

Lossless Compression Handbook

EDITOR KHALID SAYOOD



THE ACADEMIC PRESS SERIES IN COMMUNICATIONS, NETWORKING, AND MULTIMEDIA

Lossless Compression Handbook

**ACADEMIC PRESS SERIES IN COMMUNICATIONS,
NETWORKING, AND MULTIMEDIA**

Editor-in-Chief

Jerry D. Gibson

University of California, Santa Barbara

This series has been established to bring together a variety of publications that cover the latest in applications and cutting-edge research in the fields of communications and networking. The series will include professional handbooks, technical books on communications systems and networks, books on communications and network standards, research books for engineers, and tutorial treatments of technical topics for non-engineers and managers in the worldwide communications industry. The series is intended to bring the latest in communications, networking, and multimedia to the widest possible audience.

Books in the Series:

Handbook of Image and Video Processing, Al Bovik, editor

Nonlinear Image Processing, Sanjit Mitra, Giovanni Sicuranza, editors

The E-Commerce Books, Second Edition, Steffano Korper and Juanita Ellis

Multimedia Communications, Jerry D. Gibson, editor

Lossless Compression Handbook

KHALID SAYOOD, EDITOR

*Department of Electrical Engineering
University of Nebraska-Lincoln
Lincoln, Nebraska*



ACADEMIC PRESS

An imprint of Elsevier Science

AMSTERDAM / BOSTON / LONDON / NEW YORK / OXFORD / PARIS
SAN DIEGO / SAN FRANCISCO / SINGAPORE / SYDNEY / TOKYO

This book is printed on acid-free paper. ☺

Copyright © 2003 Elsevier Science (USA)

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Requests for permission to make copies of any part of the work should be mailed to: Permissions Department, Harcourt, Inc., 6277 Sea Harbor Drive, Orlando, Florida 32887-6777.

Explicit permission from Academic Press is not required to reproduce a maximum of two figures or tables from an Academic Press chapter in another scientific or research publication provided that the material has not been credited to another source and that full credit to the Academic Press chapter is given.

Academic Press

An imprint of Elsevier Science

525 B Street, Suite 1900, San Diego, California 92101-4495, USA
<http://www.academicpress.com>

Academic Press

84 Theobald's Road, London WC1X 8RR, UK
<http://www.academicpress.com>

Library of Congress Catalog Card Number: 2002104270

International Standard Book Number: 0-12-620861-1

Printed in the United States of America

02 03 04 05 06 07 9 8 7 6 5 4 3 2 1

*To Warner Miller
(1932–2001)
pioneer, mentor, and a truly decent human being*

This Page Intentionally Left Blank

Contents

List of Contributors	xvii
Preface	xix

Part I: Theory

Chapter 1: Information Theory behind Source Coding	3
<i>Frans M. J. Willems and Tjalling J. Tjalkens</i>	
1.1 Introduction	3
1.1.1 Definition of Entropy	3
1.1.2 Properties of Entropy	4
1.1.3 Entropy as an Information Measure	5
1.1.4 Joint Entropy and Conditional Entropy	6
1.1.5 Properties of Joint Entropy and Conditional Entropy	7
1.1.6 Interpretation of Conditional Entropy	8
1.2 Sequences and Information Sources	9
1.2.1 Sequences	9
1.2.2 Information Sources	9
1.2.3 Memoryless Sources	10
1.2.4 Binary Sources	10
1.2.5 Discrete Stationary Sources	10
1.2.6 The Entropy Rate	11
1.3 Variable-Length Codes for Memoryless Sources	11
1.3.1 A Source Coding System, Variable-Length Codes for Source Symbols	12
1.3.2 Unique Decodability, Prefix Codes	13
1.3.3 Kraft's Inequality for Prefix Codes and Its Counterpart	14
1.3.4 Redundancy, Entropy, and Bounds	16
1.3.5 Variable-Length Codes for Blocks of Symbols	17
1.4 Variable-Length Codes for Sources with Memory	18
1.4.1 Block Codes Again	18
1.4.2 The Elias Algorithm	19
1.4.3 Representation of Sequences by Intervals	19
1.4.4 Competitive Optimality	25

1.5	Fixed-Length Codes for Memoryless Sources, the AEP	26
1.5.1	The Fixed-Length Source Coding Problem	27
1.5.2	Some Probabilities	27
1.5.3	An Example Demonstrating the Asymptotic Equipartition Property	28
1.5.4	The Idea behind Fixed-Length Source Coding	28
1.5.5	Rate and Error Probability	28
1.5.6	A Hamming Ball	30
1.5.7	An Optimal Balance between R and P_ϵ	31
1.5.8	The Fixed-Length Coding Theorem	33
1.5.9	Converse and Conclusion	34
1.6	References	34
Chapter 2: Complexity Measures		35
<i>Stephen R. Tate</i>		
2.1	Introduction	35
2.1.1	An Aside on Computability	36
2.2	Concerns with Shannon Information Theory	37
2.2.1	Strings versus Sources	37
2.2.2	Complex Non-random Sequences	37
2.2.3	Structured Random Strings	37
2.3	Kolmogorov Complexity	38
2.3.1	Basic Definitions	39
2.3.2	Incompressibility	40
2.3.3	Prefix-free Encoding	42
2.4	Computational Issues of Kolmogorov Complexity	44
2.4.1	Resource-Bounded Kolmogorov Complexity	45
2.4.2	Lower-Bounding Kolmogorov Complexity	46
2.5	Relation to Shannon Information Theory	47
2.5.1	Approach 1: An Infinite Sequence of Sources	48
2.5.2	Approach 2: Conditional Complexities	49
2.5.3	Discussion	50
2.6	Historical Notes	51
2.7	Further Reading	51
2.8	References	51
Part II: Compression Techniques		
Chapter 3: Universal Codes		55
<i>Peter Fenwick</i>		
3.1	Compact Integer Representations	55
3.2	Characteristics of Universal Codes	55
3.3	Polynomial Representations	56
3.4	Unary Codes	57
3.5	Levenshtein and Elias Gamma Codes	58
3.6	Elias Omega and Even-Rodeh Codes	59
3.7	Rice Codes	60
3.8	Golomb Codes	62
3.9	Start-Step-Stop Codes	64

3.10	Fibonacci Codes	65
3.10.1	Zeckendorf Representation	66
3.10.2	Fraenkel and Klein Codes	66
3.10.3	Higher-Order Fibonacci Representations	66
3.10.4	Apostolico and Fraenkel Codes	67
3.10.5	A New Order-3 Fibonacci Code	69
3.11	Ternary Comma Codes	70
3.12	Summation Codes	71
3.12.1	Goldbach G_1 Codes	72
3.12.2	Additive Codes	72
3.13	Wheeler 1/2 Code and Run-Lengths	73
3.13.1	The Wheeler 1/2 Code	74
3.13.2	Using the Wheeler 1/2 Code	74
3.14	Comparison of Representations	75
3.15	Final Remarks	77
3.16	References	78

Chapter 4: Huffman Coding 79

Steven Pigeon

4.1	Introduction	79
4.2	Huffman Codes	80
4.2.1	Shannon–Fano Coding	80
4.2.2	Building Huffman Codes	80
4.2.3	N -ary Huffman Codes	83
4.2.4	Canonical Huffman Coding	84
4.2.5	Performance of Huffman Codes	84
4.3	Variations on a Theme	86
4.3.1	Modified Huffman Codes	87
4.3.2	Huffman Prefixes	87
4.3.3	Extended Huffman Codes	87
4.3.4	Length-Constrained Huffman Codes	89
4.4	Adaptive Huffman Coding	89
4.4.1	Brute Force Adaptive Huffman	89
4.4.2	The Faller, Gallager, and Knuth (FGK) Algorithm	91
4.4.3	Vitter’s Algorithm: Algorithm Λ	93
4.4.4	Other Adaptive Huffman Coding Algorithms	93
4.4.5	An Observation on Adaptive Algorithms	94
4.5	Efficient Implementations	94
4.5.1	Memory-Efficient Algorithms	95
4.5.2	Speed-Efficient Algorithms	95
4.6	Conclusion and Further Reading	97
4.7	References	97

Chapter 5: Arithmetic Coding 101

Amir Said

5.1	Introduction	101
5.2	Basic Principles	103
5.2.1	Notation	103

5.2.2	Code Values	104
5.2.3	Arithmetic Coding	106
5.2.4	Optimality of Arithmetic Coding	111
5.2.5	Arithmetic Coding Properties	112
5.3	Implementation	120
5.3.1	Coding with Fixed-Precision Arithmetic	121
5.3.2	Adaptive Coding	132
5.3.3	Complexity Analysis	142
5.3.4	Further Reading	147
5.4	References	150
Chapter 6: Dictionary-Based Data Compression: An Algorithmic Perspective		153
<i>S. Cenk Sahinalp and Nasir M. Rajpoot</i>		
6.1	Introduction	153
6.2	Dictionary Construction: Static versus Dynamic	154
6.2.1	Static Dictionary Methods	154
6.2.2	Parsing Issues	155
6.2.3	Semidynamic and Dynamic Dictionary Methods	157
6.3	Extensions of Dictionary Methods for Compressing Biomolecular Sequences	162
6.3.1	The <i>Biocompress</i> Program	162
6.3.2	The <i>GenCompress</i> Program	162
6.4	Data Structures in Dictionary Compression	163
6.4.1	Tries and Compact Tries	163
6.4.2	Suffix Trees	163
6.4.3	Trie–Reverse Trie Pairs	164
6.4.4	Karp–Rabin Fingerprints	164
6.5	Benchmark Programs and Standards	165
6.5.1	The <i>gzip</i> Program	165
6.5.2	The <i>compress</i> Program	165
6.5.3	The GIF Image Compression Standard	166
6.5.4	Modem Compression Standards: v.42bis and v.44	166
6.6	References	166
Chapter 7: Burrows–Wheeler Compression		169
<i>Peter Fenwick</i>		
7.1	Introduction	169
7.2	The Burrows–Wheeler Algorithm	170
7.3	The Burrows–Wheeler Transform	170
7.3.1	The Burrows–Wheeler Forward Transformation	170
7.3.2	The Burrows–Wheeler Reverse Transformation	171
7.3.3	Illustration of the Transformations	171
7.3.4	Algorithms for the Reverse Transformation	172
7.4	Basic Implementations	173
7.4.1	The Burrows–Wheeler Transform or Permutation	173
7.4.2	Move-To-Front Recoding	174
7.4.3	Statistical Coding	176
7.5	Relation to Other Compression Algorithms	180
7.6	Improvements to Burrows–Wheeler Compression	180

7.7	Preprocessing	181
7.8	The Permutation	181
7.8.1	Suffix Trees	183
7.9	Move-To-Front	183
7.9.1	Move-To-Front Variants	184
7.10	Statistical Compressor	185
7.11	Eliminating Move-To-Front	187
7.12	Using the Burrows–Wheeler Transform in File Synchronization	189
7.13	Final Comments	190
7.14	Recent Developments	190
7.15	References	191
Chapter 8: Symbol-Ranking and ACB Compression		195
<i>Peter Fenwick</i>		
8.1	Introduction	195
8.2	Symbol-Ranking Compression	195
8.2.1	Shannon Coder	196
8.2.2	History of Symbol-Ranking Compressors	197
8.2.3	An Example of a Symbol-Ranking Compressor	197
8.2.4	A Fast Symbol-Ranking Compressor	200
8.3	Buynovsky’s ACB Compressor	201
8.4	References	204
Part III: Applications		
Chapter 9: Lossless Image Compression		207
<i>K. P. Subbalakshmi</i>		
9.1	Introduction	207
9.2	Preliminaries	208
9.2.1	Spatial Prediction	209
9.2.2	Hierarchical Prediction	211
9.2.3	Error Modeling	212
9.2.4	Scanning Techniques	212
9.3	Prediction for Lossless Image Compression	214
9.3.1	Switched Predictors	214
9.3.2	Combined Predictors	217
9.4	Hierarchical Lossless Image Coding	220
9.5	Conclusions	222
9.6	References	223
Chapter 10: Text Compression		227
<i>Amar Mukherjee and Fauzia Awan</i>		
10.1	Introduction	227
10.2	Information Theory Background	228
10.3	Classification of Lossless Compression Algorithms	229
10.3.1	Statistical Methods	229

10.3.2	Dictionary Methods	232
10.3.3	Transform-Based Methods: The Burrows–Wheeler Transform (BWT)	233
10.3.4	Comparison of Performance of Compression Algorithms	233
10.4	Transform-Based Methods: Star (*) Transform and Length-Index Preserving Transform	234
10.4.1	Star (*) Transformation	234
10.4.2	Length-Index Preserving Transform (LIPT)	235
10.4.3	Experimental Results	237
10.4.4	Timing Performance Measurements	240
10.5	Three New Transforms—ILPT, NIT, and LIT	241
10.6	Conclusions	243
10.7	References	243
Chapter 11: Compression of Telemetry		247
<i>Sheila Horan</i>		
11.1	What is Telemetry?	247
11.2	Issues Involved in Compression of Telemetry	250
11.2.1	Why Use Compression on Telemetry	250
11.2.2	Structure of the Data	251
11.2.3	Size Requirements	251
11.3	Existing Telemetry Compression	252
11.4	Future of Telemetry Compression	253
11.5	References	253
Chapter 12: Lossless Compression of Audio Data		255
<i>Robert C. Maher</i>		
12.1	Introduction	255
12.1.1	Background	255
12.1.2	Expectations	256
12.1.3	Terminology	257
12.2	Principles of Lossless Data Compression	257
12.2.1	Basic Redundancy Removal	257
12.2.2	Amplitude Range and Segmentation	259
12.2.3	Multiple-Channel Redundancy	260
12.2.4	Prediction	260
12.2.5	Entropy Coding	262
12.2.6	Practical System Design Issues	263
12.2.7	Numerical Implementation and Portability	263
12.2.8	Segmentation and Resynchronization	263
12.2.9	Variable Bit Rate: Peak versus Average Rate	264
12.2.10	Speed and Complexity	264
12.3	Examples of Lossless Audio Data Compression Software Systems	265
12.3.1	Shorten	265
12.3.2	Meridian Lossless Packing (MLP)	266
12.3.3	Sonic Foundry Perfect Clarity Audio (PCA)	266
12.4	Conclusion	267
12.5	References	267

Chapter 13: Algorithms for Delta Compression and Remote File Synchronization 269
Torsten Suel and Nasir Memon

13.1	Introduction	269
13.1.1	Problem Definition	271
13.1.2	Content of This Chapter	271
13.2	Delta Compression	271
13.2.1	Applications	271
13.2.2	Fundamentals	273
13.2.3	LZ77-Based Delta Compressors	274
13.2.4	Some Experimental Results	275
13.2.5	Space-Constrained Delta Compression	276
13.2.6	Choosing Reference Files	278
13.3	Remote File Synchronization	279
13.3.1	Applications	279
13.3.2	The <i>rsync</i> Algorithm	280
13.3.3	Some Experimental Results for <i>rsync</i>	282
13.3.4	Theoretical Results	283
13.3.5	Results for Particular Distance Measures	285
13.3.6	Estimating File Distances	286
13.3.7	Reconciling Database Records and File Systems	286
13.4	Conclusions and Open Problems	287
13.5	References	287

Chapter 14: Compression of Unicode Files 291
Peter Fenwick

14.1	Introduction	291
14.2	Unicode Character Codings	291
14.2.1	Big-endian versus Little-endian	292
14.2.2	UTF-8 Coding	292
14.3	Compression of Unicode	293
14.3.1	Finite-Context Statistical Compressors	293
14.3.2	Unbounded-Context Statistical Compressors	293
14.3.3	LZ-77 Compressors	294
14.4	Test Compressors	294
14.4.1	The Unicode File Test Suite	294
14.5	Comparisons	295
14.6	UTF-8 Compression	296
14.7	Conclusions	296
14.8	References	297

Part IV: Standards

Chapter 15: JPEG-LS Lossless and Near Lossless Image Compression 301
Michael W. Hoffman

15.1	Lossless Image Compression and JPEG-LS	301
15.2	JPEG-LS	301

15.2.1	Overview of JPEG-LS	301
15.2.2	JPEG-LS Encoding	302
15.2.3	JPEG-LS Decoding	309
15.3	Summary	309
15.4	References	310
Chapter 16: The CCSDS Lossless Data Compression Recommendation for Space Applications		311
<i>Pen-Shu Yeh</i>		
16.1	Introduction	311
16.2	The e_Rice Algorithm	312
16.3	The Adaptive Entropy Coder	313
16.3.1	Fundamental Sequence Encoding	313
16.3.2	The Split-Sample Option	314
16.3.3	Low-Entropy Options	316
16.3.4	No Compression	317
16.3.5	Code Selection	317
16.4	Preprocessor	318
16.4.1	Predictor	318
16.4.2	Reference Sample	319
16.4.3	Prediction Error Mapper	320
16.5	Coded Data Format	321
16.6	Decoding	321
16.7	Testing	324
16.8	Implementation Issues and Applications	324
16.9	Additional Information	326
16.10	References	326
Chapter 17: Lossless Bilevel Image Compression		327
<i>Michael W. Hoffman</i>		
17.1	Bilevel Image Compression	327
17.2	JBIG	327
17.2.1	Overview of JBIG Encoding/Decoding	327
17.2.2	JBIG Encoding	330
17.2.3	Data Structure and Formatting	336
17.2.4	JBIG Decoding	338
17.3	JBIG2	339
17.3.1	Overview of JBIG2	339
17.3.2	JBIG2 Decoding Procedures	341
17.3.3	Decoding Control and Data Structures	346
17.4	Summary	348
17.5	References	349
Chapter 18: JPEG2000: Highly Scalable Image Compression		351
<i>Ali Bilgin and Michael W. Marcellin</i>		
18.1	Introduction	351
18.2	JPEG2000 Features	352
18.2.1	Compressed Domain Image Processing/Editing	353

18.2.2	Progression	353
18.3	The JPEG2000 Algorithm	354
18.3.1	Tiles and Component Transforms	354
18.3.2	The Wavelet Transform	355
18.3.3	Quantization	358
18.3.4	Bit-Plane Coding	360
18.3.5	Packets and Layers	364
18.3.6	JPEG2000 Codestream	365
18.4	Performance	366
18.5	References	369
Chapter 19: PNG Lossless Image Compression		371
<i>Greg Roelofs</i>		
19.1	Historical Background	371
19.2	Design Decisions	372
19.3	Compression Engine	374
19.4	zlib Format	376
19.5	zlib Library	376
19.6	Filters	378
19.7	Practical Compression Tips	383
19.8	Compression Tests and Comparisons	385
19.9	MNG	388
19.10	Further Reading	390
19.11	References	390
Chapter 20: Facsimile Compression		391
<i>Khalid Sayood</i>		
20.1	A Brief History	391
20.2	The Compression Algorithms	393
20.2.1	Modified Huffman	393
20.2.2	Modified READ	393
20.2.3	Context-Based Arithmetic Coding	397
20.2.4	Run-Length Color Encoding	398
20.3	The Standards	398
20.3.1	ITU-T Group 3 (T.4)	398
20.3.2	Group 4 (T.6)	399
20.3.3	JBIG and JBIG2 (T.82 and T.88)	399
20.3.4	MRC—T.44	399
20.3.5	Other Standards	402
20.4	Further Reading	402
20.5	References	402
Part V: Hardware		
Chapter 21: Hardware Implementation of Data Compression		405
<i>Sanjukta Bhanja and N. Ranganathan</i>		
21.1	Introduction	405
21.2	Text Compression Hardware	407

21.2.1	Tree-Based Encoder Example	408
21.2.2	Lempel-Ziv Encoder Example	412
21.3	Image Compression Hardware	415
21.3.1	DCT Hardware	416
21.3.2	Wavelet Architectures	416
21.3.3	JPEG Hardware	417
21.4	Video Compression Hardware	417
21.4.1	Some Detailed Examples	420
21.4.2	Commercial Video and Audio Products	426
21.5	References	442
Index		447

List of Contributors

Fauzia Awan Department of Computer Science, University of Central Florida, School of Electrical Engineering and Computer Science, Orlando, Florida 32816

Sanjukta Bhanja Department of Electrical Engineering, University of South Florida, ENB 118, 4202 East Fowler Avenue, Tampa, Florida 33620

Ali Bilgin Department of Electrical and Computer Engineering, University of Arizona, 1230 East Speedway Boulevard, Tucson, Arizona 85721, e-mail: bilgin@ece.arizona.edu

Peter Fenwick Department of Computer Science, The University of Auckland, 38 Princes Street, Room 232, Level 2, Private Bag 92019, Auckland, New Zealand, e-mail: p.fenwick@auckland.ac.nz

Michael W. Hoffman Department of Electrical Engineering, University of Nebraska-Lincoln, 209 North Walter Scott Engineering Center, Lincoln, Nebraska 68588, e-mail: mhoffman1@unl.edu

Sheila Horan Klipsch School of Electrical and Computer Engineering, New Mexico State University, Thomas and Brown Hall, Sweet and Frenger Streets, Box 30001, MSC 3-0, Las Cruces, New Mexico 88003, e-mail: Sheila@nmsu.edu

Robert C. Maher Department of Electrical and Computer Engineering, Montana State University-Bozeman, 529 Cobleigh Hall, Bozeman, Montana 59717, e-mail: rob.maher@montana.edu

Michael W. Marcellin Department of Electrical and Computer Engineering, University of Arizona, 1230 East Speedway Boulevard, Tucson, Arizona 85721, e-mail: marcellin@ece.arizona.edu

Nasir Memon CIS Department, Polytechnic University, 6 MetroTech Center, Brooklyn, New York 11201, e-mail: memon@poly.edu

Amar Mukherjee Department of Computer Science, University of Central Florida, School of Electrical Engineering and Computer Science, Orlando, Florida 32816, e-mail: amar@cs.ucf.edu

Steven Pigeon University of Montreal, 20 Yvan Blainville, Quebec, Canada J7C 1Z9, e-mail: pigeon@iro.umontreal.ca

Nasir M. Rajpoot Department of Computer Science, University of Warwick, CV4 7AL, United Kingdom

N. Ranganathan Department of Computer Science and Engineering, University of South Florida, ENB 118, Tampa, Florida 33620

Greg Roelofs PNG Development Group and Philips Semiconductors, 440 North Wolfe Road, Sunnyvale, California 94085, e-mail: newt@pobox.com

S. Cenk Sahinalp Department of Electrical Engineering and Computer Science, Case Western Reserve University, 10900 Euclid Avenue, Olin Bldg. 515, Cleveland, Ohio 44106, e-mail: cenk@cwru.edu

Amir Said Hewlett Packard Laboratories, 1501 Page Mill Road, MS 1203, Palo Alto, California 94304, e-mail: said@hpl.hp.com

Khalid Sayood Department of Electrical Engineering, University of Nebraska-Lincoln, 209 North Walter Scott Engineering Center, Lincoln, Nebraska 68588, e-mail: ksayood@ecomm.unl.edu

K. P. Subbalakshmi Department of Electrical and Computer Engineering, Stevens Institute of Technology, 208 Burchard Building, Castle Point on the Hudson, Hoboken, New Jersey 07030, e-mail: ksubbala@stevens-tech.edu

Torsten Suel Department of Computer and Information Sciences, Polytechnic University, 6 MetroTech Center, Brooklyn, New York 11201, e-mail: suel@poly.edu

Stephen R. Tate Department of Computer Science, University of North Texas, 320 General Academic Building, Mulberry and Avenue B, Denton, Texas 76203, e-mail: srt@cs.unt.edu

Tjalling J. Tjalkens Faculty of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Frans M. J. Willems Faculty of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Pen-Shu Yeh Goddard Space Flight Center, National Aeronautics and Space Administration, Code 564, Bldg. II, Rm. E215, Greenbelt, Maryland 20771, e-mail: pen-shu.yeh@gsfc.nasa.gov

Preface

Compression schemes can be divided into two major classes: lossless compression schemes and lossy compression schemes. Data compressed using lossless compression schemes can be recovered exactly, while lossy compression introduces some loss of information in the reconstruction.

While the first modern compression scheme, Huffman coding, was a lossless compression scheme, most of the initial activity in the compression area focused on lossy compression. One reason was that the “data” being considered for compression was analog—primarily samples of speech waveforms. Another reason was that the device which would be used to provide the reconstructed speech to the user, namely, the telephone, introduced so much distortion of its own that it was relatively easy to design compression schemes which introduced less distortion than that to which the user was already being subjected.

In the past two decades the situation has changed dramatically. There is a significant amount of discrete data in the form of text, graphics, images, video, and audio that needs to be stored or transmitted, and display devices are of such quality that very little distortion can be tolerated. This has resulted in a resurgence of interest in lossless compression—hence this handbook. We have tried to put together, in an accessible form, some of the most important aspects of lossless compression. The idea was to have a volume which would allow the reader to get an idea of both the depth and the breadth of the field while at the same time fulfilling the usual function of a handbook—that of being a convenient repository of information about a topic. This combination will, we hope, be useful to both the novice reader who wishes to learn more about the subject, as well as the practitioner who needs an accessible reference book.

This book consists of 21 chapters roughly divided into five sections. Each chapter is relatively self-contained. The first 2 chapters provide the theoretical underpinnings of many lossless compression schemes. These include the classical approach based on information theory and an approach fast gaining in popularity based on the ideas of Kolmogorov complexity. The chapters in the next set are devoted to well-known (and some not so well-known) methods of variable-length coding. These coding techniques are not application specific and thus have been used in a number of application areas. Chapters in the third section are devoted to particular application areas. These include text, audio, and image compression, as well as the new area of delta compression. In these chapters we describe how the various coding techniques have been used in conjunction with models which are specific to the particular application to provide lossless compression. The chapters in the fourth group describe various international standards that involve lossless compression in a variety of applications. These include standards issued by various international bodies as

well as de facto standards. The final chapter examines hardware implementations of compression algorithms.

While, of necessity, we have introduced an ordering in the chapters, there was no pedagogical intent behind the ordering. Readers can delve into any chapter independent of the other chapters. Because these chapters are self-contained, certain topics may be covered more than once. We see this as a feature rather than a bug, as the diversity of writing styles provides the reader with multiple views of these topics.

As I have read through these chapters I have been impressed by both the knowledge of the authors and by the care they have taken to make this knowledge accessible. Among these chapters are some of the best expositions on complex subjects that I have seen. I hope the readers will agree.

I am grateful to the authors for the quality of their work and for giving me the opportunity to be associated with their work. My thanks also to Joel Claypool and Lori Asbury for their help and their support through a period of some turmoil in their corner of the publishing industry.

Khalid Sayood

PART I

Theory

This Page Intentionally Left Blank

Information Theory behind Source Coding

FRANS M. J. WILLEMS
TJALLING J. TJALKENS

1.1 INTRODUCTION

Information theory forms the mathematical basis of both lossy and lossless compression. In this chapter we will look at those aspects of information which are most relevant to lossless compression.

In this section we discuss the concept of entropy. We define entropy, discuss some of its properties, and demonstrate that these properties make it a reasonable information measure. In the next sections we will show that entropy is also an information measure in a more rigorous sense. We do this by proving a “source coding theorem” for discrete memoryless sources.

1.1.1 Definition of Entropy

Assume that X is a discrete random variable that takes values in a finite alphabet \mathcal{X} . Let $|\mathcal{X}|$ denote the cardinality of the alphabet \mathcal{X} and suppose that $p(x) := \Pr\{X = x\}$ for $x \in \mathcal{X}$.

Definition 1.1. *The entropy of X is defined by*

$$H(X) := \sum_{x \in \mathcal{X}} p(x) \log \frac{1}{p(x)}. \quad (1.1)$$

There are two aspects of this definition that need further explanation. First, note that the base of the logarithm is unspecified. Usually, and throughout these chapters, we assume that this base is 2, and we say that the entropy is being measured in *bits* (from binary digits). It is also possible to assume some other base. (If we take e as the base of the logarithm, we obtain the entropy in *nats* (from natural digits).) Second, if $p(x) = 0$, then the term $p(x) \log \frac{1}{p(x)}$ in (1.1) is indeterminate; however, then we define it to be 0. This makes $H(X)$ continuous in the probability distribution, $\{p(x) : x \in \mathcal{X}\}$.

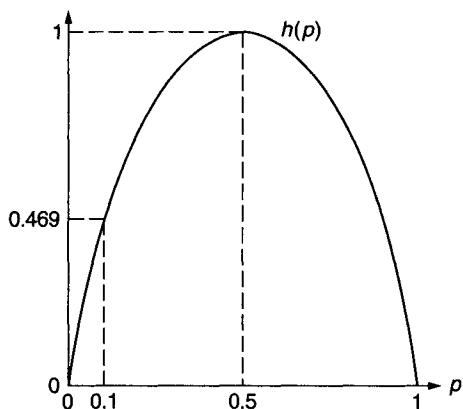


FIGURE 1.1
The binary entropy function.

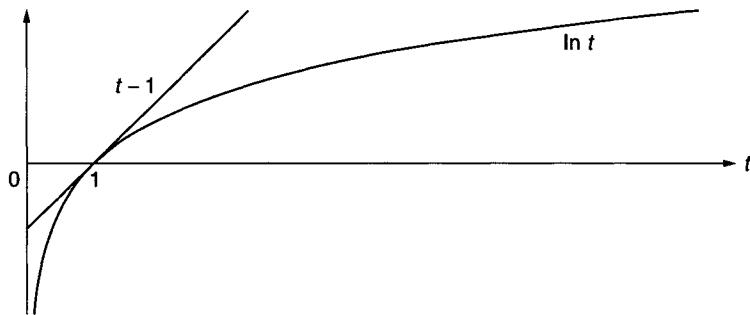


FIGURE 1.2
Geometrical illustration of the inequality $\ln t \leq t - 1$.

Example 1.1. Let X represent the outcome of a single roll with a fair die. Then $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$ and $p(x) = \frac{1}{6}$ for all $x \in \mathcal{X}$. Then, $H(X) = \log 6 = 2.585$ bits.

Example 1.2. Let $X = \{0, 1\}$ and suppose that $p(0) = 1 - p$ and $p(1) = p$ for some $0 \leq p \leq 1$. Then $H(X) = h(p)$, where $h(p) := -(1 - p)\log(1 - p) - p\log p$. The function $h(\cdot)$, which is called the *binary entropy function*, is plotted in Fig. 1.1. It can be seen that $h(0.1) = 0.469$ bits.

1.1.2 Properties of Entropy

If the random variable X takes values in the alphabet \mathcal{X} , then

Theorem 1.1. $0 \leq H(X) \leq \log |\mathcal{X}|$. Furthermore $H(X) = 0$ if and only if $p(x) = 1$ for some $x \in \mathcal{X}$, and $H(X) = \log |\mathcal{X}|$ if and only if $p(x) = \frac{1}{|\mathcal{X}|}$ for all $x \in \mathcal{X}$.

Proof. Since each $p(x)$ is ≤ 1 it follows that each term $p(x)\log \frac{1}{p(x)}$ is ≥ 0 . So $H(X) \geq 0$. Furthermore $H(X) = 0$ implies that all terms $p(x)\log \frac{1}{p(x)}$ are 0. A term is equal to 0 only if $p(x) = 0$ or $p(x) = 1$. Therefore $H(X) = 0$ only if one $p(x) = 1$ and all the rest are 0.

Before we continue with the second part of the proof we will state the perhaps most frequently used inequality in information theory, namely, $\ln t \leq t - 1$, where $t > 0$. Note that equality occurs only for $t = 1$ (see Fig. 1.2).

The proof of this inequality follows directly from the Taylor expansion around $t = 1$.

$$\ln t = \ln 1 + \frac{d}{d\tau} \ln \tau \Big|_{\tau=1} (t-1) + \frac{1}{2} \frac{d^2}{d\xi^2} \ln \xi \Big|_{\xi \in [1,t]} (t-1)^2$$

and with

$$\begin{aligned} \frac{d}{d\tau} \ln \tau \Big|_{\tau=1} &= \frac{1}{\tau} \Big|_{\tau=1} = 1, \\ \frac{d^2}{d\xi^2} \ln \xi \Big|_{\xi \in [1,t]} &= \frac{-1}{\xi^2} \Big|_{\xi \in [1,t]} < 0, \end{aligned}$$

we find

$$\ln t \leq t - 1 \quad \text{with equality only if } t = 1. \quad (1.2)$$

Now let \mathcal{X}' be the set of elements $x \in \mathcal{X}$ for which $p(x) > 0$. Using the inequality $\ln t \leq t - 1$ we then obtain

$$\begin{aligned} H(X) - \log |\mathcal{X}'| &= \sum_{x \in \mathcal{X}'} p(x) \log \frac{1}{p(x)} - \sum_{x \in \mathcal{X}'} p(x) \log |\mathcal{X}'| \\ &= \frac{1}{\ln 2} \sum_{x \in \mathcal{X}'} p(x) \ln \frac{1}{|\mathcal{X}'| p(x)} \\ &\leq \frac{1}{\ln 2} \sum_{x \in \mathcal{X}'} p(x) \left[\frac{1}{|\mathcal{X}'| p(x)} - 1 \right] \\ &= \frac{1}{\ln 2} \left[\sum_{x \in \mathcal{X}'} \frac{1}{|\mathcal{X}'|} - \sum_{x \in \mathcal{X}'} p(x) \right] = 0. \end{aligned} \quad (1.3)$$

From this and

$$\log |\mathcal{X}'| \leq \log |\mathcal{X}|, \quad (1.4)$$

it follows easily that $H(X) \leq \log |\mathcal{X}|$. Note that the two inequalities (1.3) and (1.4) hold with equality only if $p(x) = \frac{1}{|\mathcal{X}'|}$ for all $x \in \mathcal{X}$. ■

1.1.3 Entropy as an Information Measure

Information should be regarded as the entity that can resolve uncertainty. To see what this means consider a random experiment X that generates a random outcome x . Before starting this experiment we are uncertain about its actual result. The generated output x contains the information that takes away this initial uncertainty. We would like the entropy $H(X)$ to be a measure of the initial uncertainty, or equivalently, of the generated information.

Entropy as defined in (1.1) appears to have some properties that a useful measure of information must have:

- If $p(x) = 1$ for some $x \in \mathcal{X}$, then we know in advance what the output of the experiment will be. There is no initial uncertainty; in other words, the experiment generates no information. Indeed in this case the entropy $H(X)$ is equal to 0.
- If $p(x) = \frac{1}{|\mathcal{X}|}$, for all $x \in \mathcal{X}$, i.e., when all outputs are equally likely, the entropy $H(X) = \log |\mathcal{X}|$. It was pointed out by Hartley in 1928 [5] that for T equiprobable outcomes $\log T$

would be the most natural choice. The reason for this is simple. Suppose we have two random experiments that generate T_1 and T_2 equiprobable outcomes. Then the information generated by the two experiments should be equal to the information generated by the first experiment plus the information generated by the second. The logarithmic function satisfies this requirement since $\log T_1 T_2 = \log T_1 + \log T_2$.

- As we have seen in Theorem 1.1 entropy is always greater than or equal to 0 and less than or equal to $\log |\mathcal{X}|$. Indeed uncertainty can be absent, i.e., $H(X) = 0$, but it is rather difficult to think of negative uncertainty. Also it seems reasonable that only an equiprobable random experiment generates the maximal amount of information.

1.1.4 Joint Entropy and Conditional Entropy

Assume that X and Y are discrete random variables that take values in the finite alphabet \mathcal{X} and \mathcal{Y} , respectively. Suppose for $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ that

$$\begin{aligned} p(x, y) &\stackrel{\Delta}{=} \Pr\{X = x \wedge Y = y\}, \\ p(x) &\stackrel{\Delta}{=} \Pr\{X = x\} = \sum_{y \in \mathcal{Y}} \Pr\{X = x \wedge Y = y\}, \\ p(y) &\stackrel{\Delta}{=} \Pr\{Y = y\} = \sum_{x \in \mathcal{X}} \Pr\{X = x \wedge Y = y\}. \end{aligned} \quad (1.5)$$

Later we will also use

$$p(x|y) \stackrel{\Delta}{=} \Pr\{X = x | Y = y\},$$

and

$$p(y|x) \stackrel{\Delta}{=} \Pr\{Y = y | X = x\}. \quad (1.6)$$

Definition 1.2. *The joint entropy $H(X, Y)$ of a pair of discrete random variables (X, Y) with a joint distribution $p(x, y)$ is defined as*

$$H(X, Y) \stackrel{\Delta}{=} - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x, y). \quad (1.7)$$

If we consider the pair (X, Y) to be a (vector valued) random variable $Z = (X, Y)$, then this definition is a direct consequence of the entropy formula (1.1).

Definition 1.3. *The conditional entropy of X , given Y , is defined by*

$$H(X|Y) := \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{p(y)}{p(x, y)}. \quad (1.8)$$

Just like in the entropy definition we assume that $0 \log 0 = 0$. Note that $p(x) = 0$ or $p(y) = 0$ implies that $p(x, y) = 0$. Therefore terms for which $p(x, y) = 0$ do not contribute to the conditional entropy.

Example 1.3. Let $\mathcal{X} = \mathcal{Y} = \{0, 1\}$ and let $\Pr\{X = 0 \wedge Y = 0\} = \frac{3}{5}$, $\Pr\{X = 0 \wedge Y = 1\} = 0$, $\Pr\{X = 1 \wedge Y = 0\} = \frac{1}{5}$, and $\Pr\{X = 1 \wedge Y = 1\} = \frac{1}{5}$. Then $\Pr\{X = 0\} = \frac{3}{5}$, $\Pr\{X = 1\} = \frac{2}{5}$, $\Pr\{Y = 0\} = \frac{4}{5}$ and $\Pr\{Y = 1\} = \frac{1}{5}$. Now $H(X, Y) = -\frac{3}{5} \log \frac{3}{5} - 0 \log 0 - \frac{1}{5} \log \frac{1}{5} - \frac{1}{5} \log \frac{1}{5} = 1.3710$ bits and $H(X|Y) = \frac{3}{5} \log \frac{4/5}{3/5} + \frac{1}{5} \log \frac{4/5}{1/5} + \frac{1}{5} \log \frac{1/5}{1/5} = 0.649$ bits.

1.1.5 Properties of Joint Entropy and Conditional Entropy

Assume that X and Y are discrete random variables that take values in \mathcal{X} and \mathcal{Y} , respectively.

Theorem 1.2 (Chain Rule).

$$H(X, Y) = H(X) + H(Y|X). \quad (1.9)$$

Proof. In the following derivation we use the chain rule for distributions $p(x, y) = p(x)p(y|x)$.

$$\begin{aligned} H(X, Y) &= - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x, y) \\ &= - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x) \frac{p(x, y)}{p(x)} \\ &= - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x)p(y|x) \log p(x) + \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log \frac{p(x)}{p(x, y)} \\ &= - \sum_{x \in \mathcal{X}} p(x) \log p(x) + H(Y|X) \\ &= H(X) + H(Y|X). \end{aligned}$$

Theorem 1.3. *The conditional entropy $H(X|Y) \geq 0$. $H(X|Y) \leq H(X)$ with equality only if X and Y are independent.*

Proof. The conditional entropy $H(X|Y)$ is non-negative since $p(y) \geq p(x, y)$ always, so the terms in (1.8) are all non-negative.

Using $p(x) = \sum_{y \in \mathcal{Y}} p(x, y)$ we find that

$$\begin{aligned} H(X|Y) - H(X) &= \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{p(y)}{p(x, y)} - \sum_{x \in \mathcal{X}} p(x) \log \frac{1}{p(x)} \\ &= \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{p(y)}{p(x, y)} - \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{1}{p(x)} \\ &= \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{p(x)p(y)}{p(x, y)}. \end{aligned} \quad (1.10)$$

Let \mathcal{S} be the set of pairs (x, y) for which $p(x, y) > 0$. Then we continue

$$\begin{aligned} &= \frac{1}{\ln 2} \sum_{(x,y) \in \mathcal{S}} p(x, y) \ln \frac{p(x)p(y)}{p(x, y)} \\ &\leq \frac{1}{\ln 2} \sum_{(x,y) \in \mathcal{S}} p(x, y) \left[\frac{p(x)p(y)}{p(x, y)} - 1 \right] \\ &= \frac{1}{\ln 2} \left[\sum_{(x,y) \in \mathcal{S}} p(x)p(y) - \sum_{(x,y) \in \mathcal{S}} p(x, y) \right] \\ &\leq \frac{1}{\ln 2} \left[\sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x)p(y) - \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \right] = 0. \end{aligned} \quad (1.11)$$

The first inequality holds with equality only if $p(x, y) = p(x)p(y)$ for all (x, y) for which $p(x, y) > 0$. The second inequality holds with equality only if $p(x)p(y) = 0$ for (x, y) for which

$p(x, y) = 0$. Therefore $H(X) = H(X|Y)$ only if $p(x, y) = p(x)p(y)$ for all $x \in \mathcal{X}$ and all $y \in \mathcal{Y}$. ■

Example 1.4. If we compute $H(X)$ for the random variables X and Y of the previous example in this section, then we get $H(X) = \frac{3}{5} \log \frac{1}{3/5} + \frac{2}{5} \log \frac{1}{2/5} = h(\frac{2}{5}) = 0.971$ bits. Observe that $H(X) - H(X|Y) = 0.971 - 0.649 > 0$ bits.

1.1.6 Interpretation of Conditional Entropy

The conditional entropy $H(X|Y)$ can be interpreted as the (average) amount of information that X contains, after Y has been revealed. To see this, define

$$H(X|Y = y) = \sum_{x \in \mathcal{X}} p(x|y) \log \frac{1}{p(x|y)}, \quad \text{for } y \text{ such that } p(y) > 0; \quad (1.12)$$

then we can rewrite (1.8) as follows:

$$\begin{aligned} H(X|Y) &= \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y) \log \frac{p(y)}{p(x, y)} \\ &= \sum_{y \in \mathcal{Y}} p(y) \sum_{x \in \mathcal{X}} p(x|y) \log \frac{1}{p(x|y)} = \sum_{y \in \mathcal{Y}} p(y) \cdot H(X|Y = y). \end{aligned} \quad (1.13)$$

Now $H(X|Y)$ is just the average of all entropies $H(X|Y = y)$ of X for fixed y , over all $y \in \mathcal{Y}$.

Example 1.5. Again for the random pair X, Y of the previous example note that $\Pr\{X = 0|Y = 0\} = \frac{3}{4}$ and $\Pr\{X = 1|Y = 0\} = \frac{1}{4}$; therefore $H(X|Y = 0) = \frac{3}{4} \log \frac{1}{3/4} + \frac{1}{4} \log \frac{1}{1/4} = h(\frac{1}{4}) = 0.811$ bits. Furthermore $\Pr\{X = 0|Y = 1\} = 0$ and $\Pr\{X = 1|Y = 1\} = 1$; therefore $H(X|Y = 1) = 0 \log \frac{1}{0} + 1 \log \frac{1}{1} = 0$. For $H(X|Y)$ we get again $H(X|Y) = \frac{4}{5}h(\frac{1}{4}) + \frac{1}{5}0 = 0.649$ bits.

Another conditional entropy that we shall use has the form $H(X, Y|Z)$ for discrete random variables X , Y , and Z with a joint probability distribution $p(x, y, z)$. This entropy describes the amount of information in the pair (X, Y) given that Z is known (revealed). Likewise is $H(X|Y, Z)$, the amount of information in X given the pair (Y, Z) .

$$\begin{aligned} H(X, Y|Z) &= - \sum_{(x,y,z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}} p(x, y, z) \log \frac{p(x, y, z)}{p(z)} \\ &= - \sum_{z \in \mathcal{Z}} p(z) \sum_{(x,y) \in \mathcal{X} \times \mathcal{Y}} p(x, y|z) \log p(x, y|z) \\ &= \sum_{z \in \mathcal{Z}} p(z) H(X, Y|Z = z). \end{aligned} \quad (1.14)$$

$$\begin{aligned} H(X|Y, Z) &= - \sum_{(x,y,z) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Z}} p(x, y, z) \log \frac{p(x, y, z)}{p(y, z)} \\ &= - \sum_{(y,z) \in \mathcal{Y} \times \mathcal{Z}} p(y, z) \sum_{x \in \mathcal{X}} p(x|y, z) \log p(x|y, z) \\ &= - \sum_{(y,z) \in \mathcal{Y} \times \mathcal{Z}} p(y, z) H(X|Y = y, Z = z). \end{aligned} \quad (1.15)$$

The chain rule (1.2) applies and we find

$$H(X, Y|Z) = H(X|Z) + H(Y|X, Z). \quad (1.16)$$

1.2 SEQUENCES AND INFORMATION SOURCES

In this section we introduce some notation concerning sequences. We also describe different kinds of information sources.

1.2.1 Sequences

Let \mathcal{U} be a discrete alphabet. A concatenation $u = u_1 u_2 \cdots u_l$ of $l \geq 0$ symbols $u_i \in \mathcal{U}$, where $i = 1, l$ is called a *sequence over \mathcal{U}* , and l is the length of this sequence. If $l = 0$, then the sequence is called *empty*; i.e., it contains no symbols. The empty sequence is denoted by ϕ . Let \mathcal{U}^* be the set of all sequences over \mathcal{U} including the empty sequence. The length of a sequence $u \in \mathcal{U}^*$ is denoted by $\lambda(u)$.

If $u' = u'_1 u'_2 \cdots u'_{l'}^{\prime}$ and $u'' = u''_1 u''_2 \cdots u''_{l''}^{\prime\prime}$ are two sequences over \mathcal{U} , then the concatenation $u = u' u''$ of u' and u'' is defined as $u'_1 u'_2 \cdots u'_{l'} u''_1 u''_2 \cdots u''_{l''}^{\prime\prime}$. It consists of $l' + l''$ symbols. We call u' a *prefix* and u'' a *suffix* of u . Note that the empty sequence ϕ is a prefix and a suffix of all sequences.

1.2.2 Information Sources

Let time, which is denoted by t , assume the integer values $\dots, -1, 0, 1, 2, \dots$ and run from $-\infty$ to ∞ . At time t a *discrete information source* emits the source symbol x_t (see Fig. 1.3). Each symbol x_t assumes a value in the *source alphabet* \mathcal{X} . This alphabet is assumed to be *discrete*. It contains $|\mathcal{X}|$ elements and in general $|\mathcal{X}| \geq 2$. Let i and j be integers satisfying $i \leq j$. Then the concatenation $x_i^j = x_i x_{i+1} \cdots x_j$ of $j - i + 1$ source symbols generated by our source is a sequence over \mathcal{X} , called a *source sequence*.

Let T be some integer that will be specified later. It is now our intention to describe the source sequence $x_1^T = x_1 x_2 \cdots x_T$ as “efficiently” as possible. We will specify in Section 1.3.1 what we mean by efficiently here. We will often write x^j instead of x_1^j ; hence $x^j = x_1 x_2 \cdots x_j$ for $j \geq 0$. Furthermore $x^j = \phi$ for $j = 0$.

The set \mathcal{X}^T of all different sequences contains $|\mathcal{X}|^T$ elements, and the source generates the sequence $x^T \in \mathcal{X}^T$ with probability $\Pr\{X^T = x^T\}$. This probability distribution satisfies

$$\sum_{x^T \in \mathcal{X}^T} \Pr\{X^T = x^T\} = 1, \quad \text{and} \quad \Pr\{X^T = x^T\} \geq 0, \quad \text{for all } x^T \in \mathcal{X}^T. \quad (1.17)$$

We will sometimes denote $\Pr\{X^T = x^T\}$ by $P(x^T)$.

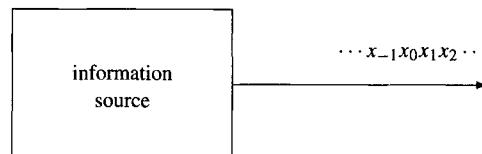


FIGURE 1.3

An information source.

1.2.3 Memoryless Sources

The simplest kind of information source that we can think of is the *memoryless* or *independent and identically distributed* (i.i.d.) source. Independence of the subsequent symbols in the sequence x^T implies that

$$\Pr\{X^T = x^T\} = \prod_{t=1,T} \Pr\{X_t = x_t\}, \quad \text{for } x^T \in \mathcal{X}^T, \quad (1.18)$$

whereas

$$\Pr\{X_t = x\} = p(x), \quad \text{for all } x \in \mathcal{X} \text{ and } t = 1, T \quad (1.19)$$

indicates that all the symbols in the source sequence are identically distributed. The symbol distribution $\{p(x) : x \in \mathcal{X}\}$ satisfies $\sum_{x \in \mathcal{X}} p(x) = 1$ and $p(x) \geq 0$ for all $x \in \mathcal{X}$. In Sections 1.3 and 1.5 we will focus on memoryless sources.

1.2.4 Binary Sources

A *binary* source has source alphabet $\mathcal{X} = \{0, 1\}$ and $|\mathcal{X}| = 2$. The source sequence x^T is now a sequence of binary digits of length T . For a memoryless binary source we have that

$$1 - p(0) = p(1) \stackrel{\Delta}{=} \theta, \quad (1.20)$$

and the statistical behavior of a binary i.i.d. source is therefore completely described by a single parameter θ that specifies the probability that the source generates a “1”. So if we consider a sequence x^T with a zeros and $b = T - a$ ones, the probability $\Pr\{X^T = x^T\} = (1 - \theta)^a \theta^b$.

1.2.5 Discrete Stationary Sources

There exists a more general class of sources where the successive source outputs are not independent. In that case we know that due to Theorem 1.3 the entropy of the sequence X^T must be smaller than T times the single-letter entropy. In order to understand these sources it is useful to study the behavior of the entropy for sequences of increasing length.

First we shall discuss briefly the class of sources known as the *stationary sources*. Stationary sources produce infinite-length output strings $\dots, X_{-2}, X_{-1}, X_0, X_1, X_2, \dots$ and assign probabilities to arbitrary source output sequences, with the following two restrictions.

1. The probabilities must be *consistent*. This means that if $J = \{i_1, i_2, \dots, i_n\}$ is a set of n indices, for arbitrary n , and j is an index that is not in J , then for all possible symbol values x_k , for $k = j$ or $k \in J$, the following holds:

$$\begin{aligned} & \Pr\{X_{i_1} = x_{i_1} \wedge X_{i_2} = x_{i_2} \wedge \dots \wedge X_{i_n} = x_{i_n}\} \\ &= \sum_{x_j \in \mathcal{X}} \Pr\{X_{i_1} = x_{i_1} \wedge X_{i_2} = x_{i_2} \wedge \dots \wedge X_{i_n} = x_{i_n} \wedge X_j = x_j\}. \end{aligned}$$

Example 1.6. An example will clarify this condition. Suppose $\mathcal{X} = \{0, 1, 2\}$ and the source assigns $\Pr\{X_1 = 0 \wedge X_2 = 0\} = \frac{1}{2}$, $\Pr\{X_1 = 0 \wedge X_2 = 1\} = \frac{1}{8}$, and $\Pr\{X_1 = 0 \wedge X_2 = 2\} = \frac{1}{8}$. Then by summing over $X_2 \in \mathcal{X}$ we find $\Pr\{X_1 = 0\} = \frac{3}{4}$. If the source does not assign this probability to $\Pr\{X_1 = 0\}$, it is not consistent.

2. The probabilities depend only on their relative positions, not on the absolute positions. So they are *shift invariant*. If again J is a set of n indices, then

$$\begin{aligned} & \Pr \{X_{i_1} = x_{i_1} \wedge X_{i_2} = x_{i_2} \wedge \cdots \wedge X_{i_n} = x_{i_n}\} \\ &= \Pr \{X_{i_1+1} = x_{i_1} \wedge X_{i_2+1} = x_{i_2} \wedge \cdots \wedge X_{i_n+1} = x_{i_n}\}. \end{aligned}$$

The entropy $H(X^T)$ is usually an unbounded non-decreasing function of T because it is reasonable to assume that for many sources most source outputs contribute to the entropy; i.e., that for an infinite number of source outputs X_i the conditional entropy $H(X_i|X^{i-1})$ is strictly positive. For this reason it is more useful to study the *averaged* per letter entropy defined by

$$H_T(X) \triangleq \frac{1}{T} H(X^T). \quad (1.21)$$

1.2.6 The Entropy Rate

We shall now consider the behavior of $H_T(X)$ as T grows. First note that for a memoryless source with entropy $H(X)$ we have

$$H_T(X) = \frac{1}{T} H(X^T) = \frac{1}{T} T H(X) = H(X). \quad (1.22)$$

So, in this case the *averaged* per letter entropy is indeed the actual per letter entropy.

As it will turn out, the conditional entropy $H(X_T|X^{T-1})$, also known as the *innovation entropy*, plays the same role as $H_T(X)$. The following theorem will describe the behavior of these two entropies.

Theorem 1.4. *For a stationary source $\{X_i\}_{i=-\infty}^{\infty}$ with $H(X_1) < \infty$ holds*

1. $H(X_T|X^{T-1})$ is non-increasing in T .
2. $H_T(X) \geq H(X_T|X^{T-1})$ for all $T = 1, 2, 3, \dots$.
3. $H_T(X)$ is non-increasing in T .
4. $\lim_{T \rightarrow \infty} H_T(X) = \lim_{T \rightarrow \infty} H(X_L|X^{T-1})$.

For a proof of this theorem we refer to [2] or [3] where the interested reader can also find a more extended discussion of sources with memory.

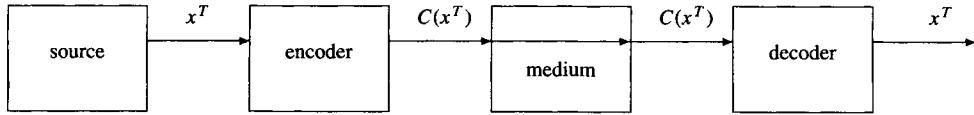
Definition 1.4. *The entropy rate $H_{\infty}(X)$ is defined by*

$$H_{\infty}(X) \triangleq \lim_{T \rightarrow \infty} H_T(X). \quad (1.23)$$

It is possible to prove a source coding theorem that shows that the entropy rate plays the same role for stationary sources as $H(X)$ does for memoryless sources; i.e., it describes the smallest possible code rate for data compression.

1.3 VARIABLE-LENGTH CODES FOR MEMORYLESS SOURCES

In this section we describe the elements in a source coding system and discuss what we want to achieve with such a system. First we investigate codes for single source symbols. We argue that, in order to achieve unique decodability, we can apply prefix codes. We derive the Kraft inequality for prefix codes and use this inequality to show that the expected codeword length of a prefix code cannot be smaller than the entropy of the source. Moreover we show that there exist prefix

**FIGURE 1.4**

A source coding system.

codes with an expected codeword length smaller than the source entropy plus 1. Finally we show that codes for large blocks of source symbols achieve an expected codeword length per source symbol arbitrary close to source entropy.

1.3.1 A Source Coding System, Variable-Length Codes for Source Symbols

Suppose that we have a memoryless source with alphabet \mathcal{X} and symbol distribution $\{p(x) : x \in \mathcal{X}\}$. We now want to transmit the sequence $x^T = x_1 x_2 \cdots x_T$ generated by the source via a binary medium (or store it into a binary medium). Therefore we must transform the source sequence x^T into a sequence $C(x^T)$ of binary¹ digits called a code sequence from which the original source sequence x^T can be reconstructed. The length of the code sequence is denoted $L(x^T)$. Our notation suggests that $C(\cdot)$ is a mapping from \mathcal{X}^T into the set $\{0, 1\}^*$ of binary sequences of variable lengths. The transformation from a source sequence into a code sequence is performed by an *encoder*; the reverse operation is carried out by the *decoder* (see Fig. 1.4).

We assume that both the encoder and decoder are operating *sequentially*; i.e., the encoder first accepts the first source symbol x_1 of the source sequence x^T as an input, then the second symbol x_2 , etc. Using these source symbols it first produces the first binary code digit c_1 of the code sequence $C(x^T)$, then the second binary digit c_2 , etc. The first binary digit c_1 in the code sequence $C(x^T)$ is the first input that is applied to the decoder, and the second binary digit c_2 of this code sequence is its second input, etc. The decoder first reconstructs from these code digits the first source symbol x_1 , after that the second symbol x_2 , etc.

To use the medium efficiently it is important that the transformation is such that the length of the code sequence $C(x^T)$ is *as short as possible*.

In this chapter we only consider transformations of the following kind. We assume that there corresponds a variable-length codeword $c(x) \in \{0, 1\}^*$ with length $l(x)$ to each source symbol $x \in \mathcal{X}$. The mapping $c(\cdot)$ from source symbols in \mathcal{X} into codewords from $\{0, 1\}^*$ is called a *source code*. The code sequence $C(x^T)$ is now the concatenation of the codewords $c(x_t)$ corresponding to all the source symbols x_t for $t = 1, T$; i.e.,

$$C(x^T) = c(x_1)c(x_2)\cdots c(x_T). \quad (1.24)$$

If $L(x^T)$ denotes the length of the code sequence $C(x^T)$, then, by the ergodic theorem (see, e.g., [4] or [11]), we know that with probability 1

$$\lim_{T \rightarrow \infty} \frac{L(x^T)}{T} = \lim_{T \rightarrow \infty} \frac{l(x_1) + l(x_2) + \cdots + l(x_T)}{T} = \sum_{x \in \mathcal{X}} p_x l(x) = \mathbf{E}[l(X)]. \quad (1.25)$$

Therefore, to achieve short code sequences, we must choose the source code $\{c(x) : x \in \mathcal{X}\}$, such that the expected codeword length $\mathbf{E}[l(X)]$ is as small as possible.

¹ In this chapter we will consider only the case where all code symbols are binary.

Table 1.1

x	$c_1(x)$	$c_2(x)$	$c_3(x)$	$c_4(x)$
a	0	0	00	0
b	0	1	10	10
c	1	00	11	110
d	10	11	110	111

However, to make it possible for the decoder to reconstruct the original source sequence from the code sequence the source code $\{c(x) : x \in \mathcal{X}\}$ also must be *uniquely decodable*. This will be the subject of the next section.

1.3.2 Unique Decodability, Prefix Codes

We start this section with an example.

Example 1.7. Suppose that the source alphabet $\mathcal{X} = \{a, b, c, d\}$. Table 1.1 gives four possible source codes, $c_1(\cdot)$, $c_2(\cdot)$, $c_3(\cdot)$, and $c_4(\cdot)$, for this alphabet. We assume that all symbols $x \in \mathcal{X}$ have positive probability so that there are no codewords for symbols that never occur. Note that we use these codes to encode a sequence of T source symbols.

Note that code $c_1(\cdot)$ has the property that there are two source symbols (a and b) that are represented by the same codeword (0) and the decoder will be confused when it receives this codeword.

The codewords in the second code are all distinct. However, this code is still not good for encoding a *sequence* of source symbols; e.g., the source sequences ac and ca both give the same codeword (000) and again the decoder is confused when receiving this code sequence.

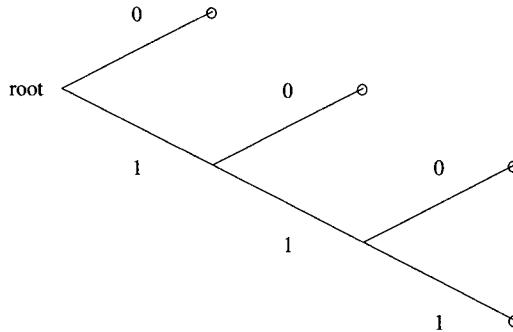
To see how the third code acts, assume that a sequence of codewords is received. The decoder starts from the beginning. If the first two bits in the sequence of codewords are 00 or 10, the first source symbol is a or b , respectively. However, if the first two bits are 11, then the following bits must also be checked. If the third bit is a 1, then the first source symbol is c . If the third bit is 0, the length of the string of 0's starting with this third bit determines the source symbol. An even length can result only from symbol c ; an odd length implies that the first codeword is 110 and thus the source symbol is d . Note that eventually after having seen many bits in the sequence of codewords we will be able to determine the first source symbol. Therefore this third code is *uniquely decodable*.

The fourth code is uniquely decodable too but is also *instantaneous*. Observe that no codeword in this code is the prefix of any other codeword in the code. Therefore we call this code a *prefix code*. Now, if the decoder has received a sequence so far which is a codeword, it can immediately output the corresponding source symbol since no other codeword has the received sequence as a prefix.

In this chapter, instead of considering all uniquely decodable codes, we will consider only prefix codes. The reason behind this choice will be given at the end of this section. First we give a definition.

Definition 1.5. A code is called a *prefix code* or an *instantaneous code* if no codeword is the prefix of any other codeword.

Codewords in a prefix code can be regarded as leaves in a code tree; see Fig. 1.5. Using this tree it is easy to parse a code sequence uniquely into the codewords; e.g., 00101111011... is parsed into 0, 0, 10, 111, 10, 11,

**FIGURE 1.5**

The code tree corresponding to a code with codewords 0, 10, 110, and 111. The path from the root to a leaf determines the codeword.

Note that prefix codes are uniquely decodable (but not all uniquely decodable codes are prefix codes). If we now return to our original problem, describing source sequences efficiently, we know from (1.25) that we must minimize the expected codeword length $E[l(X)]$. If we restrict ourselves to prefix codes, we must find out what the smallest possible $E[l(X)]$ is that can be achieved with prefix codes.

The third code in the example did not satisfy the prefix condition. Nevertheless it was possible to decode this code as we have seen. The third code is *uniquely decodable*. These codes are treated extensively by Cover and Thomas [2, Chapter 5]. Here we mention only that every uniquely decodable code can be changed into a prefix code without changing the codeword lengths $l(x)$, $x \in \mathcal{X}$. This implies that, in general, there is no good reason to use codes other than prefix codes.

1.3.3 Kraft's Inequality for Prefix Codes and Its Counterpart

In order to find the smallest possible expected codeword length $E[l(X)]$ that can be achieved with prefix codes, the Kraft inequality is of crucial importance.

Theorem 1.5. *The codeword lengths $l(x)$ of any binary prefix code for the source symbols $x \in \mathcal{X}$ must satisfy Kraft's inequality, which says that*

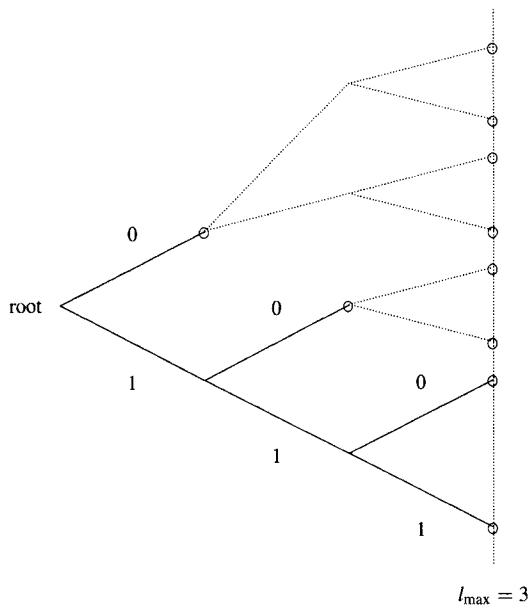
$$\sum_{x \in \mathcal{X}} 2^{-l(x)} \leq 1. \quad (1.26)$$

There is also a Kraft inequality for non-binary code alphabets. For a D -ary prefix code the inequality $\sum_{x \in \mathcal{X}} D^{-l(x)} \leq 1$ holds. We consider only binary codes here.

Proof. Consider Fig. 1.6. Let $l_{\max} = \max_{x \in \mathcal{X}} l(x)$. In the tree representing the code, a codeword $c(x)$ with length $l(x)$ has $2^{l_{\max}-l(x)}$ descendants at level l_{\max} . Because of the prefix condition no codeword is an ancestor of any other codeword in the tree. Therefore the descendants at level l_{\max} of all codewords must be distinct and there are $2^{l_{\max}}$ nodes at level l_{\max} ; hence

$$\sum_{x \in \mathcal{X}} 2^{l_{\max}-l(x)} \leq 2^{l_{\max}}. \quad (1.27)$$

Dividing both sides of (1.27) by $2^{l_{\max}}$ completes the proof. ■

**FIGURE 1.6**

The prefix code with codewords 0, 10, 110, and 111 and their descendants at level $l_{\max} = 3$.

Example 1.8. For the prefix code with codewords 0, 10, 110, and 111 we get the Kraft sum

$$\sum_{x \in \mathcal{X}} 2^{-l(x)} = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1. \quad (1.28)$$

This Kraft sum satisfies the Kraft inequality with equality.

In the Kraft theorem (1.5) we have shown that codeword lengths in any prefix code satisfy Kraft's inequality. There is also a counterpart to the Kraft theorem. This is our next result.

Theorem 1.6. *If the codeword lengths $l(x) \in \mathcal{X}$ for symbols $x \in \mathcal{X}$ satisfy Kraft's inequality (1.26), there exists a binary prefix code with codewords $c(x)$ that have length $l(x)$.*

Proof. Assume without loss of generality that $\mathcal{X} = \{1, 2, \dots, M\}$ with $M \geq 2$ and that

$$l(1) \leq l(2) \leq \dots \leq l(M). \quad (1.29)$$

We start with a full tree with depth $l_{\max} = \max_{x \in \mathcal{X}} l(x)$; this tree has $2^{l_{\max}}$ leaves. The first codeword $c(1)$ always can be placed in the tree. Next suppose that $m < M$ codewords have already been placed in the tree. Then

$$\sum_{x=1}^m 2^{-l(x)} + \sum_{x=m+1}^M 2^{-l(x)} \leq 1. \quad (1.30)$$

The second term in (1.30) is positive; hence the first term is strictly less than 1. This first term is a multiple of $2^{-l(m)}$ and is therefore less than or equal to $1 - 2^{-l(m)}$, so there still exists a node at level $l(m)$. Therefore a codeword $c(m+1)$ with length $l(m+1) \geq l(m)$ definitely fits in the tree. Continuing like this, all codewords get their place in the tree. Note that the path from the root of the tree to a codeword node determines the codeword. ■

Kraft's inequality and its counterpart play an important role in the analysis and design of codes for (memoryless) sources as we will see in the following sections.

1.3.4 Redundancy, Entropy, and Bounds

Remember that we are still discussing variable-length codes for symbols generated by memoryless sources. The source distribution is $\{p(x) : x \in \mathcal{X}\}$ and a code is described by $\{c(x) : x \in \mathcal{X}\}$.

Definition 1.6. We define the individual redundancy $\rho(x)$ of the symbol $x \in \mathcal{X}$, whose codeword $c(x)$ has length $l(x)$ as

$$\rho(x) \triangleq l(x) - \log_2 \frac{1}{p(x)}, \quad (1.31)$$

and the expected redundancy of a code as

$$\mathbf{E}[\rho(X)] \triangleq \sum_{x \in \mathcal{X}} p(x)\rho(x). \quad (1.32)$$

Redundancies are measured in bits.²

The term $\log_2(1/p(x))$ in the definition of individual redundancy is called the *ideal codeword length*. The average ideal codeword length is equal to the source entropy $H(X)$.

For the expected redundancy we can show that

$$\begin{aligned} \mathbf{E}[\rho(X)] &= \sum_{x \in \mathcal{X}} p(x)\rho(x) \\ &= \sum_{x \in \mathcal{X}} p(x)l(x) - \sum_{x \in \mathcal{X}} p(x)\log_2 \frac{1}{p(x)} = \mathbf{E}[l(X)] - H(X). \end{aligned} \quad (1.33)$$

If we take

$$l(x) = \left\lceil \log_2 \frac{1}{p(x)} \right\rceil \quad \text{for all possible } x \text{ with } p(x) > 0, \quad (1.34)$$

where $\lceil a \rceil$ is the smallest integer larger than a , then

$$\sum_{x \in \mathcal{X}} 2^{-l(x)} = \sum_{x \in \mathcal{X}} 2^{-\lceil \log_2 \frac{1}{p(x)} \rceil} \leq \sum_{x \in \mathcal{X}} 2^{-\log_2 \frac{1}{p(x)}} = \sum_{x \in \mathcal{X}} p(x) = 1; \quad (1.35)$$

hence there exists a prefix code with codeword lengths $l(x) = \lceil \log_2(1/p(x)) \rceil$ for all possible x with $p(x) > 0$ by Theorem 1.6. For this codeword length assignment we obtain the following upper bound for the individual redundancy:

$$\rho(x) = l(x) - \log_2 \frac{1}{p(x)} = \left\lceil \log_2 \frac{1}{p(x)} \right\rceil - \log_2 \frac{1}{p(x)} < 1. \quad (1.36)$$

Note that this bounds holds for all symbols $x \in \mathcal{X}$ with $p(x) > 0$. Averaging over $x \in \mathcal{X}$ immediately leads to the following statement:

² Logarithms have base 2, here and in the next chapters.

Theorem 1.7. *For symbols generated by a source according to the probability distribution $\{p(x) : x \in \mathcal{X}\}$, there exists a prefix code that achieves*

$$\mathbf{E}[\rho(X)] < 1, \quad \text{or equivalently} \quad \mathbf{E}[l(X)] < H(X) + 1. \quad (1.37)$$

This theorem tells us that codes with expected codeword length $\mathbf{E}[l(X)]$ less than $H(X) + 1$ exist. An obvious question is now: “Can we find better codes than those described by 1.34?” The next theorem gives at least a partial answer to this question. It says that there can exist only slightly better codes.

Theorem 1.8. *For any prefix code for the source symbols generated by a source with probability distribution $\{p(x) : x \in \mathcal{X}\}$,*

$$\mathbf{E}[\rho(X)] \geq 0, \quad \text{or equivalently,} \quad \mathbf{E}[l(X)] \geq H(X), \quad (1.38)$$

where equality occurs if and only if $l(x) = \log_2(1/p(x))$, i.e., if the codeword lengths are equal to the ideal codeword lengths, for all $x \in \mathcal{X}$.

Proof. Consider the following chain of (in)equalities. The summations are only over symbols $x \in \mathcal{X}$ with $p(x) > 0$. We now obtain

$$\begin{aligned} \mathbf{E}[\rho(X)] &= \sum_{x \in \mathcal{X}} p(x)l(x) - \sum_{x \in \mathcal{X}} p(x) \log_2 \frac{1}{p(x)} \\ &= -\frac{1}{\ln 2} \sum_{x \in \mathcal{X}} p(x) \ln \frac{2^{-l(x)}}{p(x)} \\ &\geq -\frac{1}{\ln 2} \sum_{x \in \mathcal{X}} p(x) \left(\frac{2^{-l(x)}}{p(x)} - 1 \right) = \frac{1}{\ln 2} \left(\sum_{x \in \mathcal{X}} p(x) - \sum_{x \in \mathcal{X}} 2^{-l(x)} \right) \geq 0. \end{aligned} \quad (1.39)$$

In this proof the first inequality holds since $\ln a \leq a - 1$ for $a > 0$. The second inequality is a consequence of Kraft’s inequality, which holds for prefix codes (see Theorem 1.5). Note (see Fig. 1.2) that equality can occur only if $a = 1$ or equivalently $l(x) = \log_2(1/p(x))$ for all $x \in \mathcal{X}$. ■

From Theorem 1.7 and Theorem 1.8 we can conclude that for a source with entropy $H(X)$ there exists a binary prefix code such that

$$H(X) \leq \mathbf{E}[l(X)] < H(X) + 1. \quad (1.40)$$

1.3.5 Variable-Length Codes for Blocks of Symbols

In Section 1.3.1 we described a system in which each source symbol $x_t, t = 1, 2, \dots$ was encoded separately. We have investigated the properties of such a system and we could show (see (1.40)) that there exists a prefix code with expected codeword length $\mathbf{E}[l(X)]$ not more than one binary digit per source symbol larger than the symbol entropy $H(X)$. For small symbol entropies this is not a very exciting result. A better result can be obtained by *blocking*. In a blocking system, blocks of T source symbols are encoded together into a single codeword from a prefix code.

Consider a prefix code that maps source sequence $x^T = x_1 x_2 \cdots x_T$ onto codeword $C(x^T)$ having length $L(x^T)$ for all $x^T \in \mathcal{X}^T$. Then it is not a big surprise that

$$\mathbf{E}[L(X^T)] \geq H(X^T), \quad (1.41)$$

for any prefix code for source blocks of length T . However, there exists a prefix code that achieves

$$\mathbf{E}[L(X^T)] < H(X^T) + 1. \quad (1.42)$$

For memoryless sources, see Theorems 1.2 and 1.3.

$$H(X^T) = H(X_1 X_2 \cdots X_T) = H(X_1) + H(X_2) + \cdots + H(X_T) = T H(X). \quad (1.43)$$

Therefore the expected code sequence length per source symbol for any prefix code for a block of T source symbols satisfies

$$\frac{\mathbf{E}[L(X^T)]}{T} \geq H(X), \quad (1.44)$$

but there exists a prefix code that achieves

$$\frac{\mathbf{E}[L(X^T)]}{T} < H(X) + \frac{1}{T}. \quad (1.45)$$

Consequently

$$\lim_{T \rightarrow \infty} \frac{\mathbf{E}[L(X^T)]}{T} = H(X); \quad (1.46)$$

in other words, $H(X)$ is the smallest possible number of codebits needed to describe a source symbol of a source with symbol entropy. This demonstrates the importance of the notion of entropy.

1.4 VARIABLE-LENGTH CODES FOR SOURCES WITH MEMORY

In this section we consider codes for source with memory. First we consider codes for blocks of source symbols. The practical value of these codes is not very large. Therefore, we also discuss the Elias algorithm, which is the basis of all arithmetic codes. An arithmetic encoder computes the code sequence from the source sequence and the corresponding decoder computes the source sequence from the code sequence. We demonstrate that the performance need not be much worse than that of any prefix code for a block of source symbols.

1.4.1 Block Codes Again

For sources with memory we can also use blocking to increase the efficiency of the coded representation. In a blocking system, blocks of T source symbols are encoded together into a single codeword from a prefix code. Consider a prefix code that maps source sequence $x^T = x_1 x_2 \cdots x_T$ onto codeword $C(x^T)$ having length $L(x^T)$ for all $x^T \in \mathcal{X}^T$. Then it is not a big surprise that

$$\mathbf{E}[L(X^T)] \geq H(X^T), \quad (1.47)$$

for any prefix code for source blocks of length T . However, there exists a prefix code that achieves

$$\mathbf{E}[L(X^T)] < H(X^T) + 1. \quad (1.48)$$

Again, as in (1.44)–(1.46) we consider the expected code sequence length per source symbol and obtain

$$\lim_{T \rightarrow \infty} \frac{\mathbf{E}[L(X^T)]}{T} = H_\infty(X). \quad (1.49)$$

From this we conclude that in general we should prefer large values of T .

Note, however, that it is a huge job to design and implement a prefix code for larger values of T . For example, for binary sources and $T = 30$ we get more than a billion codewords! A better approach is described in the next section.

1.4.2 The Elias Algorithm

1.4.2.1 Introduction to the Elias Algorithm

An *arithmetic encoder* computes the code sequence $C(x^T)$ that corresponds to the actual source sequence x^T . The corresponding decoder reconstructs the actual source sequence from this code sequence again by performing simple computations. Such an approach differs from encoding and decoding using tables. These tables need to be constructed first and then stored in memory. Arithmetic coding does not require construction and storage of the complete code before the encoding and decoding process is begun. On the other hand, we see that arithmetic coding requires some computational effort during encoding and decoding.

Using arithmetic codes it is possible to process long source sequences. Large values of the length T reduce the redundancy per source symbol. All arithmetic codes are based on the Elias algorithm, which was unpublished but described by Abramson [1] and Jelinek [6].

The first idea behind the Elias algorithm is that to each source sequence x^T there corresponds a *subinterval* of the unit interval $[0, 1]$. This principle can be traced back to Shannon [10].

The second idea, due to Elias, is to order the sequences x^t of length t *lexicographically*, for $t = 1, \dots, T$. For two sequences x^t and \tilde{x}^t we have that $x^t < \tilde{x}^t$ if and only if there exists a $\tau \in \{1, 2, \dots, t\}$ such that $x_i = \tilde{x}_i$ for $i = 1, 2, \dots, \tau - 1$ and $x_\tau < \tilde{x}_\tau$. This lexicographical ordering makes it possible to compute the subinterval corresponding to a source sequence sequentially (recursively).

To make things simple we assume in this section that the source alphabet is binary; hence $\mathcal{X} = \{0, 1\}$.

1.4.3 Representation of Sequences by Intervals

Let distribution $P_c(x^t)$, $x^t \in \{0, 1\}^t$, $t = 0, 1, \dots, T$ be what we shall call the *coding distribution*. It is known to both encoder and decoder. We require that

$$\begin{aligned} P_c(\phi) &= 1, \\ P_c(x^{t-1}) &= P_c(x^{t-1}, X_t = 0) + P_c(x^{t-1}, X_t = 1), \quad \text{for all } x^{t-1} \in \{0, 1\}^{t-1}, t = 1, \dots, T, \\ P_c(x^T) &> 0 \quad \text{for all } x^T \in \{0, 1\}^T \text{ with } \Pr[X^T = x^T] > 0. \end{aligned} \tag{1.50}$$

To each source sequence there now corresponds a subinterval of $[0, 1]$.

Definition 1.7. *The interval $I(x^t)$ corresponding to $x^t \in \{0, 1\}^t$, $t = 0, 1, \dots, T$ is defined as*

$$I(x^t) \stackrel{\Delta}{=} [B(x^t), B(x^t) + P_c(x^t)], \tag{1.51}$$

where $B(x^t) \stackrel{\Delta}{=} \sum_{\tilde{x}^t < x^t} P_c(\tilde{x}^t)$.

Note that for $t = 0$ we have that $P_c(\phi) = 1$ and $B(\phi) = 0$ (the only sequence of length 0 is ϕ itself), and consequently $I(\phi) = [0, 1]$. Observe that for any fixed value of t , $t = 0, 1, \dots, T$, all intervals $I(x^t)$ are disjoint, and their union is $[0, 1]$. Each interval has a length equal to the corresponding coding probability.

Just like all source sequences, each code sequence $C = c_1 \cdots c_L$ can be associated with a subinterval of $[0, 1)$.

Definition 1.8. *The interval $J(c_1 \cdots c_L)$ corresponding to the code sequence $c_1 \cdots c_L$ is defined as*

$$J(c_1 \cdots c_L) \triangleq [F(c_1 \cdots c_L), F(c_1 \cdots c_L) + 2^{-L}), \quad (1.52)$$

with $F(c_1 \cdots c_L) \triangleq \sum_{l=1,L} c_l 2^{-l}$.

To understand this, note that the code sequence $c_1 \cdots c_L$ can be considered as a binary fraction $F(c_1 \cdots c_L)$. Since $c_1 \cdots c_L$ is followed by other code sequences, the decoder receives a stream of code digits from which only the first L digits correspond to $c_1 \cdots c_L$. The decoder can determine the value that is represented by the binary fraction formed by the total stream $c_1 c_2 \cdots c_L c_{L+1} \cdots$; i.e.,

$$F_\infty \triangleq \sum_{l=1,\infty} c_l 2^{-l}, \quad (1.53)$$

where it should be noted that the length of the total stream is not necessarily infinite. Since $F(c_1 \cdots c_L) \leq F_\infty < F(c_1 \cdots c_L) + 2^{-L}$ we may say that the interval $J(c_1 \cdots c_L)$ corresponds to the code sequence $c_1 \cdots c_L$.

1.4.3.1 Compression

To compress a sequence x^T , we search for a short code sequence $c_1 \cdots c_L$ whose code interval $J(c_1 \cdots c_L)$ is contained in the sequence interval $I(x^T)$. This is the main principle in arithmetic coding.

Definition 1.9. *The codeword $c_1 \cdots c_L$ for source sequence x^T consists of*

$$L \triangleq \left\lceil \log_2 \frac{1}{P_c(x^T)} \right\rceil + 1 \quad (1.54)$$

binary digits such that

$$F(c_1 \cdots c_L) \triangleq \lceil B(x^T) \cdot 2^L \rceil \cdot 2^{-L}. \quad (1.55)$$

Note that we need to consider only sequences x^T with $\Pr\{X^T = x^T\} > 0$. They have $P_c(x^T) > 0$.

Since

$$F(c_1 \cdots c_L) \geq B(x^T) \quad (1.56)$$

and

$$F(c_1 \cdots c_L) + 2^{-L} < B(x^T) + 2^{-L} + 2^{-L} \leq B(x^T) + P_c(x^T), \quad (1.57)$$

we may conclude that $J(c_1 \cdots c_L) \subseteq I(x^T)$, and therefore $F_\infty \in I(x^T)$. Since the intervals $I(x^T)$ for $x^T \in \{0, 1\}^T$ are all disjoint, the decoder can reconstruct the source sequence x^T from F_∞ . Note that after this reconstruction, the decoder can compute the code sequence $c_1 \cdots c_L$, just like the encoder, and find the location of the first digit of the *next* code sequence. Note also that, since all code intervals are disjoint, no code sequence is the prefix of any other code sequence. Thus the Elias code satisfies the prefix condition. From (1.55) we obtain the following result:

Theorem 1.9. *The arithmetic code that we have just described achieves code sequence lengths $L(x^T)$ that satisfy*

$$L(x^T) < \log_2 \frac{1}{P_c(x^T)} + 2, \quad (1.58)$$

for all $x^T \in \{0, 1\}^T$ with $\Pr\{X^T = x^T\} > 0$.

Note that the difference between the code sequence length $L(x^T)$ and $\log(1/P_c(x^T))$ is always less than 2 bits. We say that the *individual coding redundancy* is less than 2 bits. We immediately see that this results in an upper bound of 2 bits for the expected redundancy.

1.4.3.2 Sequential Computation

The lexicographical ordering over the source sequences makes it possible to compute the interval $I(x^T)$ sequentially. To do so, we transform the starting interval $I(\phi) = [0, 1)$ into $I(x_1)$, $I(x_1x_2)$, ..., and $I(x_1x_2 \cdots x_T)$, respectively. The consequence of the lexicographical ordering over the source sequences is that

$$\begin{aligned} B(x^t) &= \sum_{\tilde{x}^t < x^t} P_c(x^t) \\ &= \sum_{\tilde{x}^{t-1} < x^{t-1}} P_c(\tilde{x}^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x^{t-1}, \tilde{x}_t) \\ &= B(x^{t-1}) + \sum_{\tilde{x}_t < x_t} P_c(x^{t-1}, \tilde{x}_t). \end{aligned} \quad (1.59)$$

In other words $B(x^t)$ can be computed from $B(x^{t-1})$ and $P_c(x^{t-1}, X_t = 0)$. Therefore the encoder and the decoder can easily find $I(x^t)$ after having determined $I(x^{t-1})$, if it is “easy” to determine probabilities $P_c(x^{t-1}, X_t = 0)$ and $P_c(x^{t-1}, X_t = 1)$ after having processed $x_1x_2 \cdots x_{t-1}$. Observe that when the symbol x_t is being processed, the source interval $I(x^{t-1}) = [B(x^{t-1}), B(x^{t-1}) + P_c(x^{t-1})]$ is subdivided into two subintervals

$$I(x^{t-1}, X_t = 0) = [B(x^{t-1}), B(x^{t-1}) + P_c(x^{t-1}, X_t = 0))$$

and

$$I(x^{t-1}, X_t = 1) = [B(x^{t-1}) + P_c(x^{t-1}, X_t = 0), B(x^{t-1}) + P_c(x^{t-1})]. \quad (1.60)$$

The encoder proceeds with one of these subintervals depending on the actual symbol x_t ; therefore $I(x^t) \subseteq I(x^{t-1})$. This implies that

$$I(x^T) \subseteq I(x^{T-1}) \subseteq \cdots \subseteq I(\phi). \quad (1.61)$$

The decoder determines from F_∞ the source symbols x_1, x_2, \dots, x_T , respectively, by comparing F_∞ with thresholds $D(x^{t-1})$.

Definition 1.10. *The thresholds $D(x^{t-1})$ are defined as*

$$D(x^{t-1}) \triangleq B(x^{t-1}) + P_c(x^{t-1}, X_t = 0), \quad (1.62)$$

for $t = 1, 2, \dots, T$.

Observe that threshold $D(x^{t-1})$ splits up the interval $I(x^{t-1})$ in two parts (see (1.60)). It is the upper boundary point of $I(x^{t-1}, X_t = 0)$ but also the lower boundary point of $I(x^{t-1}, X_t = 1)$.

Since $F_\infty \in I(x^T) \subseteq I(x^t)$, we have for $D(x^{t-1})$ that

$$\begin{aligned} F_\infty &< B(x^t) + P_c(x^t) \\ &= B(x^{t-1}) + P_c(x^{t-1}, X_t = 0) \\ &= D(x^{t-1}) \quad \text{if } x_t = 0, \end{aligned} \tag{1.63}$$

and

$$\begin{aligned} F_\infty &\geq B(x^t) \\ &= B(x^{t-1}) + P_c(x^{t-1}, X_t = 0) \\ &= D(x^{t-1}) \quad \text{if } x_t = 1. \end{aligned} \tag{1.64}$$

Therefore the decoder can easily find x_t by comparing F_∞ with the threshold $D(x^{t-1})$. Consequently the decoder can also operate sequentially.

We conclude this section with the observation that the Elias algorithm combines an acceptable coding redundancy with a desirable *sequential* implementation. The number of operations is *linear* in the source sequence length T . It is *crucial*, however, that the encoder and decoder have access to or can easily determine the probabilities $P_c(x^{t-1}, X_t = 0)$ and $P_c(x^{t-1}, X_t = 1)$ after having processed $x_1 x_2 \dots x_{t-1}$. If we accept a loss of at most 2 bits of coding redundancy, we are left with the problem of finding good, sequentially available, coding distributions $P_c(x^T)$.

Example 1.9. Assume that $T = 3$ and the source emits the sequence $x^T = 011$. The coding distribution is as in Fig. 1.7 in which (some of) the conditional coding probabilities are shown.

An arithmetic encoder now computes the codeword for $x^T = 011$ as follows. From $B(\phi) = 0$ the encoder first computes $B(0) = B(\phi) = 0$ and $P_c(0) = 0.7$. Then $B(01) = B(0) + P_c(00) = 0 + P_c(0) \cdot 0.5 = 0.7 \cdot 0.5 = 0.35$ and $P_c(01) = P_c(0) \cdot 0.5 = 0.7 \cdot 0.5 = 0.35$. Finally $B(011) = B(01) + P_c(010) = 0.35 + P_c(01) \cdot 0.9 = 0.35 + 0.35 \cdot 0.9 = 0.665$ and $P_c(011) = P_c(01) \cdot 0.1 = 0.35 \cdot 0.1 = 0.035$. Now the codeword can be determined. The length $L = \lceil \log(1/0.035) \rceil + 1 = 6$ and therefore c^L is the binary expansion of $[0.665 \cdot 64]/64 = 43/64$, which is 101011. See Fig. 1.8.

To see how the decoder operates note that $\frac{43}{64} \leq F_\infty < \frac{44}{64}$. First the decoder compares F_∞ with $D(\phi) = B(\phi) + P_c(0) = 0 + 0.7 = 0.7$. Clearly $F_\infty < \frac{44}{64} < 0.7$ and therefore the decoder finds $x_1 = 0$. Just like the encoder, the decoder computes $B(0)$ and $P_c(0)$. The next threshold is $D(0) = B(0) + P_c(00) = 0 + P_c(0) \cdot 0.5 = 0.7 \cdot 0.5 = 0.35$. Now $F_\infty \geq \frac{43}{64} > 0.35$ and $x_2 = 1$. The decoder computes $B(01)$ and $P_c(01)$ and the next threshold $D(01) = B(01) + P_c(010) = 0.35 + P_c(01) \cdot 0.9 = 0.35 + 0.35 \cdot 0.9 = 0.665$. Again $F_\infty \geq \frac{43}{64} > 0.665$ and the decoder produces $x_3 = 1$. After computation of $B(011)$ and $P_c(011)$ the decoder can reconstruct the

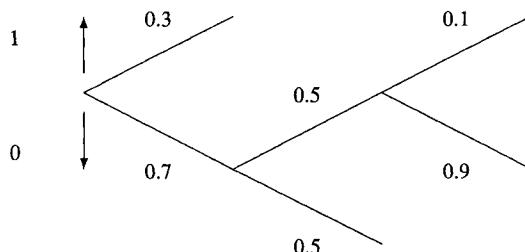
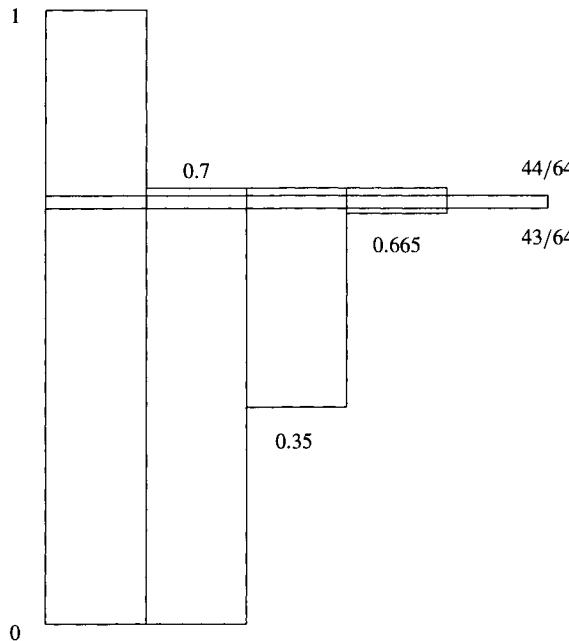


FIGURE 1.7

A graph with some conditional coding probabilities, e.g., $P_c(X_1 = 0) = 0.7$, $P_c(X_2 = 1|X_1 = 0) = 0.5$, and $P_c(X_3 = 1|X_1 = 0, X_2 = 1) = 0.1$.

**FIGURE 1.8**

Subintervals corresponding to the coding of 011.

codeword 101011. Note that for this example the coding redundancy is $6 - \log(1/0.035) = 1.163$ bits.

1.4.3.3 Achieving Entropy

In the previous subsections we have seen that code sequence lengths approximately equal to $-\log_2 P_c(x^T)$ are achievable if we use the Elias algorithm. The question now is how the coding distribution should be chosen. We consider only the case here when the probability distribution over the source sequences is known. In that case we can take

$$P_c(x^T) = \Pr\{X^T = x^T\} = P(x^T), \quad \text{for all } x^T \in \mathcal{X}^T. \quad (1.65)$$

What we achieve for each source sequence x^T with $P(x^T) > 0$ by doing so is

$$L(x^T) < \log_2 \frac{1}{P_c(x^T)} + 2 = \log_2 \frac{1}{P(x^T)} + 2; \quad (1.66)$$

hence the code sequence length is not more than 2 digits larger than the *ideal code sequence length*. Therefore the following bound for the expected codeword length is obtained:

$$\begin{aligned} \mathbb{E}[L(X^T)] &= \sum_{x^T \in \mathcal{X}^T} P(x^T)L(x^T) \\ &< \sum_{x^T \in \mathcal{X}^T} P(x^T) \left(\log_2 \frac{1}{P(x^T)} + 2 \right) = H(X^T) + 2. \end{aligned} \quad (1.67)$$

If we compare upper bound (1.67) with upper bound (1.48) we see that the difference is only one binary digit. Note that the lower bound (1.47) also holds for expected code sequence length corresponding to the Elias method.

Theorem 1.10. *The Elias method yields a prefix code with*

$$H(X^T) \leq \mathbf{E}[L(X^T)] < H(X^T) + 2. \quad (1.68)$$

This is achieved by taking the actual probability distribution of the source sequences as coding distribution.

1.4.3.4 A Special Case

We assume that the source probabilities $P(x^T)$ satisfy a special condition, namely, that for all sequences x^T holds

$$P(x^T) = 2^{-i(x^T)}, \quad \text{for positive integers } i(x^T). \quad (1.69)$$

Such a probability distribution is called a *dyadic distribution*.

The first consequence of a dyadic distribution is that $-\log_2 P(x^T)$ is an integer and thus the upward rounding in (1.54) is not necessary.

If we also assume that the probabilities are ordered in non-increasing order, then $B(x^T) \cdot \frac{1}{P(x^T)}$ is a non-negative integer. This is easy to see if we start from the definition of $B(x^T)$ just below (1.51).

Let $\ell = -\log_2 P(x^T)$, so ℓ is a positive integer. We write

$$\begin{aligned} B(x^T) \cdot 2^\ell &= \sum_{\tilde{x}^T < x^T} P(\tilde{x}^T) 2^\ell, \\ &= \sum_{\tilde{x}^T < x^T} 2^{-i(\tilde{x}^T)} 2^\ell, \\ &= \sum_{\tilde{x}^T < x^T} 2^{\ell - i(\tilde{x}^T)}. \end{aligned} \quad (1.70)$$

Because the probabilities are ordered in non-increasing order we know that for all \tilde{x}^T with $\tilde{x}^T < x^T$, $\ell - i(\tilde{x}^T) \geq 0$ and so $B(x^T) \cdot 2^\ell$ is a sum of positive integers. This implies that the upward rounding in (1.55) is not needed and we find

$$F(c_1 \cdots c_L) = B(x^T), \quad (1.71)$$

$$F(c_1 \cdots c_L) + 2^{-\ell} = B(x^T) + P(x^T). \quad (1.72)$$

So we can conclude that the code sequences with lengths $-\log_2 P(x^T)$ form a prefix code. Clearly the expected code sequence length of this code equals the entropy $H(X^T)$ and thus this code realizes the Shannon lower bound.

Note that if the source is memoryless, the lexicographical order that is required by the Elias algorithm does not satisfy the above non-increasing order property.

In universal source coding problems the actual probability distribution of the source sequences is unknown. Then clever choices of the coding distribution will lead to expected code sequence lengths that are not much larger than what we have found here.

1.4.3.5 Accuracy Issues

At the end of this section we mention that we have avoided discussing accuracy issues. These issues were first investigated by Rissanen [9], who has contributed a lot more to arithmetic coding, and (independently) by Pasco [8]. More details and practical implementations of the arithmetic coding algorithm can be found in Chapter 5.

1.4.4 Competitive Optimality

The Elias algorithm realizes an expected code sequence length close to the minimal expected code sequence length. This of course does not imply that the Elias algorithm gives the shortest code sequence lengths for all source sequences x^T . A better question to ask is how likely is it that another code produces code sequences that are (much) smaller than those of the Elias code.

With the code sequence lengths from (1.54) we can find the following result.

Theorem 1.11. *Let $L(x^T)$ be the code sequence length of the Elias code and let $L'(x^T)$ be the code sequence length of any other (decodable) code. Also, let c be an arbitrary positive constant; then*

$$\Pr\{L(X^T) \geq L'(X^T) + c\} \leq 2^{-c+2}. \quad (1.73)$$

Proof.

$$\Pr\{L(X^T) \geq L'(X^T) + c\} = \Pr\{\lceil -\log_2 P(X^T) \rceil \geq L'(X^T) + c - 1\} \quad (1.74)$$

$$\leq \Pr\{-\log_2 P(X^T) \geq L'(X^T) + c - 2\} \quad (1.75)$$

$$\leq \Pr\{P(X^T) \leq 2^{-L'(X^T)-c+2}\} \quad (1.76)$$

$$= \sum_{x^T: P(x^T) \leq 2^{-L'(x^T)-c+2}} P(x^T) \quad (1.77)$$

$$\leq \sum_{x^T: P(x^T) \leq 2^{-L'(x^T)-c+2}} 2^{-L'(x^T)-c+2} \quad (1.78)$$

$$\leq 2^{-c+2} \sum_{x^T} 2^{-L'(x^T)} \quad (1.79)$$

$$\leq 2^{-c+2}, \quad (1.80)$$

where in the last step we used Kraft's inequality. ■

Because the code sequence length $L(x^T)$ can become much larger than c by choosing T large enough, we may conclude that the Elias code cannot be much worse than any other code most of the time.

An even stronger result can be obtained in the special case where the probability distribution is ordered in non-increasing order and is dyadic; see the definition in (1.69). First we shall define a property that we are interested in.

Definition 1.11. *A code with code sequence lengths $L(x^T)$ is said to be competitively optimal with respect to a given source (probability distribution) if for all other uniquely decodable codes with lengths $L'(x^T)$ the following holds:*

$$\Pr\{L(X^T) < L'(X^T)\} \geq \Pr\{L(X^T) > L'(X^T)\}. \quad (1.81)$$

Note that the optimality with respect to *expected code sequence length* does not automatically imply competitive optimality.

In the case of the Elias code for dyadic and non-increasingly ordered probability distributions we know that $L(x^T) = -\log_2 P(x^T)$. The result we now obtain will be stated in the following theorem.

Theorem 1.12 (Cover and Thomas). *For a dyadic and non-increasingly ordered probability distribution $P(x^T)$ the Elias code has code sequence lengths $L(x^T) = -\log_2 P(x^T)$. Let $L'(x^T)$*

be the lengths of any other uniquely decodable code for this source. Then

$$\Pr\{L(X^T) < L'(X^T)\} \geq \Pr\{L(X^T) > L'(X^T)\}, \quad (1.82)$$

with equality if and only if $L'(x^T) = L(x^T)$ for all x^T .

Thus the code whose lengths satisfy the condition $L(x^T) = -\log_2 P(x^T)$ is the *unique* competitively optimal code.

Proof. Consider the function $\text{sgn}(x)$ defined as

$$\text{sgn}(x) = \begin{cases} 1; & \text{if } x > 0 \\ 0; & \text{if } x = 0 \\ -1; & \text{if } x < 0. \end{cases} \quad (1.83)$$

It is easy to check that

$$\text{sgn}(x) \leq 2^x - 1, \quad \text{for all } x \text{ with } x \leq 0 \text{ or } x \geq 1. \quad (1.84)$$

Equality holds only when $x = 0$ or $x = 1$.

Consider the difference

$$\begin{aligned} \Pr\{L(X^T) > L'(X^T)\} - \Pr\{L(X^T) < L'(X^T)\} &= \sum_{x^T: L'(x^T) < L(x^T)} P(x^T) - \sum_{x^T: L'(x^T) > L(x^T)} P(x^T) \\ &= \sum_{x^T} P(x^T) \text{sgn}(L(x^T) - L'(x^T)) \\ &\leq \sum_{x^T} 2^{-L(x^T)} (2^{L(x^T) - L'(x^T)} - 1) \\ &= \sum_{x^T} 2^{-L'(x^T)} - \sum_{x^T} 2^{-L(x^T)} \\ &\leq 0. \end{aligned} \quad (1.85)$$

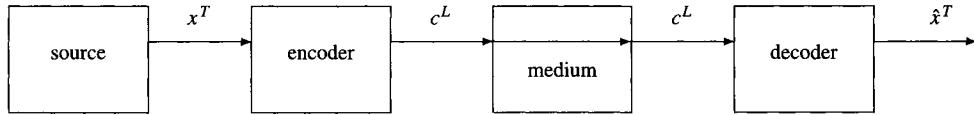
In this derivation we used (1.84) and, in the last step, we used the Kraft inequality twice, which holds with equality for the lengths $L(x^T)$.

Equality in the theorem holds if the two inequalities in the derivation above hold with equality. For the first one this requires that for all x^T we have either $L(x^T) - L'(x^T) = 0$ or $L(x^T) - L'(x^T) = 1$. The second requirement implies that the Kraft inequality must hold with equality for $L'(x^T)$ also and together this implies that $L'(x^T) = L(x^T)$ for all x^T . ■

The results in this section are a slightly different variation of the results presented by Cover and Thomas [2, Section 5.11].

1.5 FIXED-LENGTH CODES FOR MEMORYLESS SOURCES, THE AEP

In this section we will pursue a different approach to source coding. It is based on the AEP. We want the code sequences to have a fixed length that is as short as possible. If the length of the code sequences is smaller than the source sequence length, there do not exist code sequences for all source sequences and errors can occur. We show here that the error probability can be made

**FIGURE 1.9**

A source coding system.

arbitrarily small only if the number of code symbols per source symbol is larger than the source entropy.

1.5.1 The Fixed-Length Source Coding Problem

Consider the source coding situation in Fig. 1.9. There a source generates a sequence $x^T = x_1 x_2 \cdots x_T$ consisting of T source symbols. For reasons of simplicity we assume in this chapter that the source is binary; hence $\mathcal{X} = \{0, 1\}$. The encoder observes the source sequence x^T and produces the fixed-length code sequence $c^L = c_1 \cdots c_L$ that corresponds to sequence x^T , so $c^L = f_e(x^T)$. The sequence c^L consists of L binary digits and is conveyed to the decoder. The decoder produces the sequence \hat{x}^T that is represented by the codeword c^L ; hence $\hat{x}^T = f_d(c^L)$.

In general the decoder will not always produce the correct estimate \hat{x}^T of the actual source sequence x^T . Hence $\hat{x}^T = x^T$ will not always hold. Therefore the *error probability* P_ϵ , which is defined by

$$P_\epsilon \triangleq \Pr\{\hat{X}^T \neq X^T\}, \quad (1.86)$$

will not be zero. It is our problem in this chapter is to find an encoder and a decoder that achieve a desirable balance between minimizing the code sequence length L and minimizing the error probability P_ϵ . The solution to this problem depends strongly on the *asymptotic equipartition property*. We will develop this concept in the next sections.

1.5.2 Some Probabilities

Consider a binary memoryless source with parameter $\theta = \Pr\{X_t = 1\}$ for $t = 1, \dots, T$. The probability that this source generates a *specific* sequence $x_1 x_2 \cdots x_T$ that contains e ones (and hence $T - e$ zeros) is $\theta^e (1 - \theta)^{T-e}$. There are $\binom{T}{e} = T! / ((T - e)! e!)$ sequences containing e ones and $T - e$ zeros. Therefore the probability that the source produces an arbitrary sequence with e ones (and hence $T - e$ zeros) is equal to

$$\Pr\{E = e\} = \binom{T}{e} \theta^e (1 - \theta)^{T-e}. \quad (1.87)$$

Here E is the random variable whose value is the number of ones in the sequence $x_1 x_2 \cdots x_T$.

Example 1.10. For parameter $\theta = 1/5$ and for source sequence length $T = 5$ the probability that a specific sequence with 2 ones, e.g., 10001, occurs is $(1/5)^2 (4/5)^3 = 64/3125$. The number of sequences of length 5 with 2 ones is $\binom{5}{2} = \frac{5!}{2!(3!)^2} = 10$; thus the probability that an arbitrary sequence of length 5 containing 2 ones occurs is

$$\Pr\{E = 2\} = \binom{5}{2} (1/5)^2 (4/5)^3 = 128/625. \quad (1.88)$$

1.5.3 An Example Demonstrating the Asymptotic Equipartition Property

Example 1.11. Again assume that the source parameter $\theta = 0.2$. For $T = 10, 100$, and 1000 we have computed the probabilities $\Pr\{E = e\}$ for $e = 0, 1, \dots, T$. The results for $T = 10$, $T = 100$, and $T = 1000$ are shown in Fig. 1.10. Observe that the distribution of E concentrates around $0.2T$ when the sequence length T increases. This is a consequence of the law of large numbers.

Next we compute for sequence lengths $T = 10, 100, 1000$, (a) the probability $\Pr\{E \leq 0.25T\}$ that a source with parameter $\theta = 0.2$ produces a sequence with not more than $0.25T$ ones and (b) the number of sequences $\sum_{e \leq 0.25T} \binom{T}{e}$ containing not more than $0.25T$ ones. These probabilities and numbers can be found in Table 1.2. Observe that (a) the probability $\Pr\{E \leq 0.25T\}$ approaches 1 for $T \rightarrow \infty$ and (b) the number $\sum_{e \leq 0.25T} \binom{T}{e}$ is always considerably less than 2^T , the total number of sequences of length T . In the next sections we will make all this more precise.

1.5.4 The Idea behind Fixed-Length Source Coding

The idea behind source coding is to reserve binary code sequences of length L only for sequences x^T in a properly chosen subset \mathcal{A} of $\{0, 1\}^T$ (see Fig. 1.11). Hence there are only $|\mathcal{A}|$ code sequences. The set \mathcal{A} , if it is well chosen, is often called the *set of typical sequences* or *typical set*. For the length L of the code sequences we can now write

$$L = \lceil \log_2 |\mathcal{A}| \rceil. \quad (1.89)$$

If the source produces a sequence $x^T \notin \mathcal{A}$ one of the code sequences for source sequences in \mathcal{A} is used but this will certainly lead to an error; hence

$$P_\epsilon = \Pr\{X^T \notin \mathcal{A}\}. \quad (1.90)$$

Example 1.12. Let $T = 10$ and assume that the set \mathcal{A} contains all sequences with 0, 1, or 2 ones; hence $|\mathcal{A}| = 1 + \binom{10}{1} + \binom{10}{2} = 56$. We need code sequence length $L = \lceil \log_2 56 \rceil = 6$ to obtain unique code sequences for all source sequences in \mathcal{A} ; see Fig. 1.12. Note that an error occurs only if $x^T \notin \mathcal{A}$. In that case the code sequence for source sequence 0000000000 is used.

1.5.5 Rate and Error Probability

The performance of a fixed-length source coding system is determined by the *rate* R and the *error probability* P_ϵ . The rate is defined as

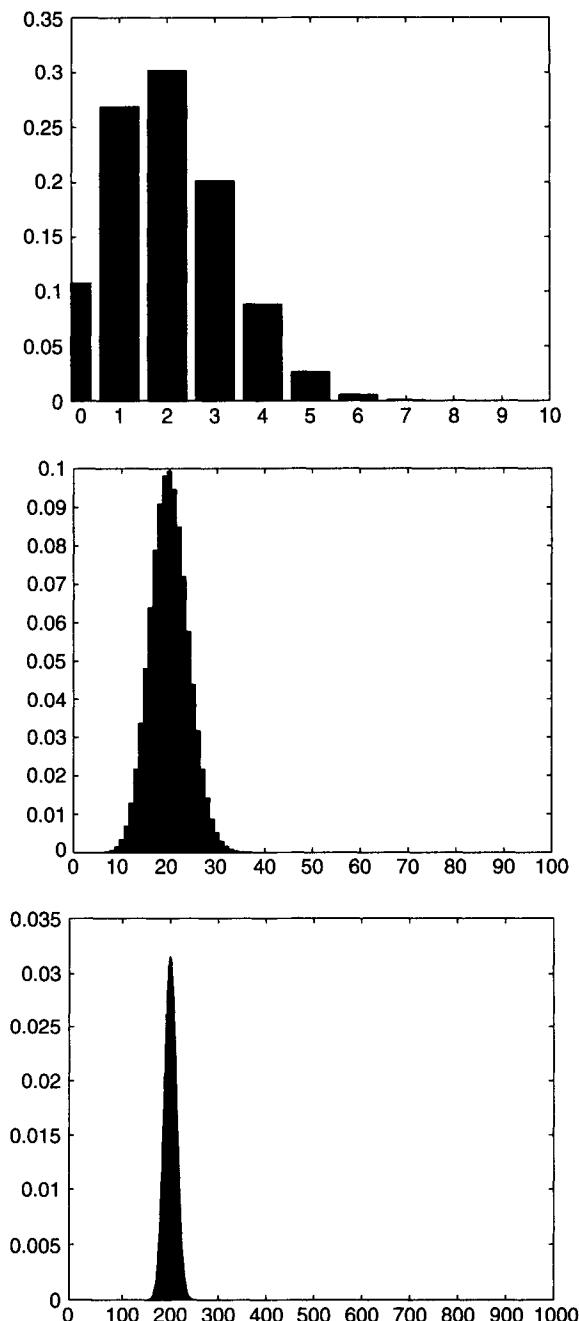
$$R \triangleq \frac{L}{T} = \frac{\lceil \log_2 |\mathcal{A}| \rceil}{T}. \quad (1.91)$$

It specifies how many binary code digits per source symbol are necessary. For the error probability we can write

$$P_\epsilon = \Pr\{\hat{X}^T \neq X^T\} = \Pr\{X^T \notin \mathcal{A}\}. \quad (1.92)$$

We must now choose the set \mathcal{A} such that R is small without making P_ϵ large.

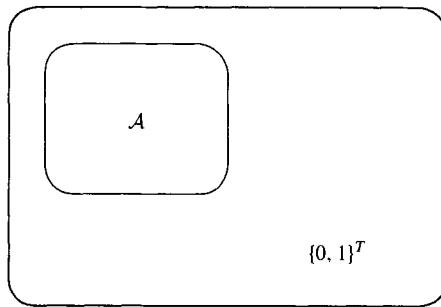
Example 1.13. For $T = 10$ and set \mathcal{A} that contains all sequences with 0, 1, or 2 ones, the rate and error probability are $R = \frac{\lceil \log_2 56 \rceil}{10} = \frac{\lceil 5.8074 \rceil}{10} = 0.6$ and $P_\epsilon = \Pr\{X^T \notin \mathcal{A}\} = 0.3222$, respectively.

**FIGURE 1.10**

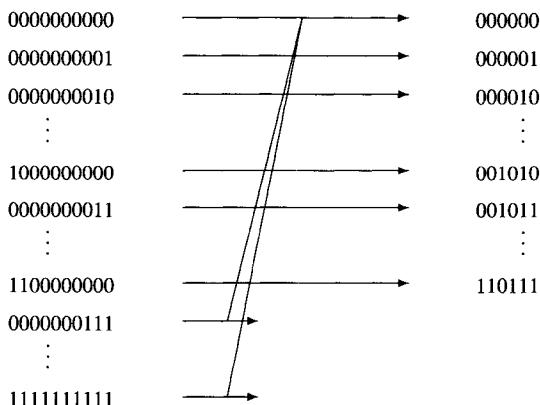
Probabilities $\Pr\{E = e\}$ for $e = 0, 1, \dots, T$ and $t = 10, 100$, and 1000.

Table 1.2

T	$\Pr\{E \leq 0.25T\}$	$\sum_{e \leq 0.25T} \binom{T}{e}$
10	0.6778	$56 = 2^{5.8074}$
100	0.9125	$2^{78.241}$
1000	0.9999	$2^{806.76}$

**FIGURE 1.11**

The typical set \mathcal{A} , a subset of $\{0, 1\}^T$.

**FIGURE 1.12**

Source sequences of length $T = 10$ and the corresponding code sequences.

1.5.6 A Hamming Ball

Suppose that we can afford M code sequences. This results in a rate $R = \lceil \log_2 M \rceil / T$. We then achieve the smallest possible error probability P_ϵ when we choose for the set \mathcal{A} the M *most probable* source sequences. Assume that $0 \leq \theta < \frac{1}{2}$. If we now rank the source sequences according to their probability, the “first” sequence is $000 \cdots 0$. Then “follow” the sequences with a single one, and then the sequences with 2 ones, etc. The “last” sequence $111 \cdots 1$ is the least probable sequence. The typical sets \mathcal{A}_δ for $0 \leq \delta \leq 1$ are therefore defined as

$$\mathcal{A}_\delta \triangleq \{x^T : w_H(x^T) \leq \delta T\} \quad (1.93)$$

and are therefore optimal; they achieve the smallest possible P_ϵ over all sets with the same size. The Hamming weight $w_H(x^T)$ of a sequence x^T is defined as the number of non-zero symbols in this sequence. We call a set \mathcal{A}_δ a *Hamming ball*.

1.5.7 An Optimal Balance between R and P_ϵ

Assume that the set \mathcal{A} , i.e., the set of source sequences that have their own code sequence, is the Hamming ball \mathcal{A}_δ . We can now find an upper bound on the rate R and on the error probability P_ϵ as a function of δ , for $\delta > \theta$. Therefore we must define two functions first. Note that the binary entropy function was used in Example 1.2 and given in Fig. 1.1. It is repeated here for convenience.

Definition 1.12. Suppose that $0 \leq \delta \leq 1$. First we define the binary entropy function as

$$h(\delta) \triangleq \delta \log_2 \frac{1}{\delta} + (1 - \delta) \log_2 \frac{1}{1 - \delta}. \quad (1.94)$$

If moreover we assume that $0 < \theta < 1$, we can define the binary divergence function as

$$d(\delta||\theta) \triangleq \delta \log_2 \frac{\delta}{\theta} + (1 - \delta) \log_2 \frac{1 - \delta}{1 - \theta}. \quad (1.95)$$

Example 1.14. Plots of $h(\delta)$ and $d(\delta||\theta)$ are shown in Fig. 1.13. Observe that the entropy function $h(\delta)$ is a non-negative increasing convex- \cap function of δ for $0 \leq \delta \leq \frac{1}{2}$. The divergence function $d(\delta||\theta)$ is related to the tangent to the entropy function $h(\delta)$ in $(\theta, h(\theta))$. This tangent is described by $r_\theta(\delta) = \delta \log_2 \frac{1}{\theta} + (1 - \delta) \log_2 \frac{1}{1 - \theta}$. Now the divergence function $d(\delta||\theta) = r_\theta(\delta) - h(\delta)$. In Fig. 1.13 the value $\theta = 0.2$.

We now can formulate the following result.

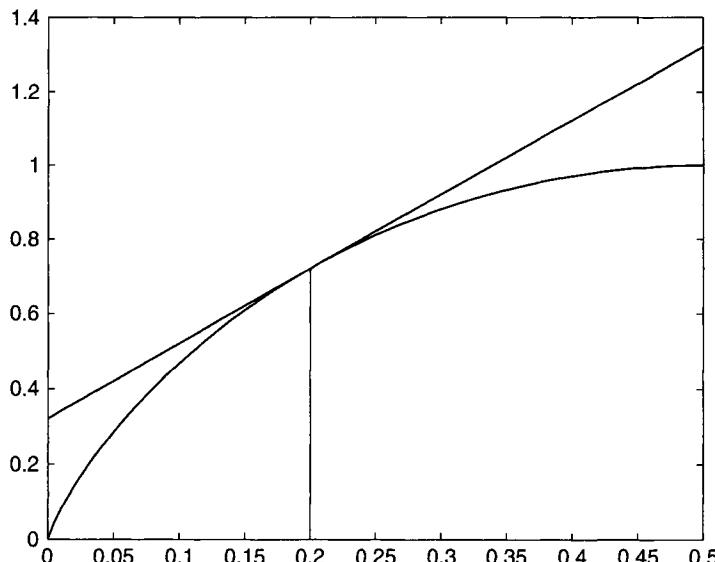


FIGURE 1.13

The binary entropy function $h(\delta)$ and its tangent $r_\theta(\delta)$ in $(\theta, h(\theta))$.

Theorem 1.13. *For a binary memoryless source with parameter θ consider fixed-length source codes that assign code sequences only to the source sequences in a Hamming ball \mathcal{A}_δ . These codes achieve, for $0 < \theta < \delta \leq \frac{1}{2}$ and source sequence length T , rates and error probabilities that satisfy*

$$R = \frac{\lceil \log_2 |\mathcal{A}_\delta| \rceil}{T} \leq h(\delta) + \frac{1}{T}, \quad (1.96)$$

$$P_\epsilon = \Pr\{X^T \notin \mathcal{A}_\delta\} \leq 2^{-Td(\delta||\theta)}. \quad (1.97)$$

Note that for a source with $\theta = 0$ we can easily achieve $R = 0$ and at the same time $P_\epsilon = 0$. Achieving $R = 1$ together with $P_\epsilon = 0$ is also straightforward.

Before we prove this theorem we give an example.

Example 1.15. We can compare the bounds (1.96) and (1.97) to the actual rates R and error probabilities P_ϵ . With $\theta = 0.2$ and $\delta = 0.25$ we first determine

$$h(\delta) = h(0.25) = 0.8113$$

$$d(\delta||\theta) = d(0.25||0.2) = 0.0106.$$

This gives us, for source sequence lengths $T = 10, 100$ and 1000 , the following upper bounds on the rates and error probabilities:

T	R	$h(\delta) + \frac{1}{T}$	P_ϵ	$2^{-Td(\delta \theta)}$
10	0.6	0.9113	0.3222	0.9288
100	0.79	0.8213	0.0875	0.4780
1000	0.807	0.8123	5.07×10^{-5}	6.22×10^{-4}

The error probabilities and rates in this table follow from Table 1.2.

Proof. (A) We will first upper bound the volume of a Hamming ball. Let $0 \leq \delta \leq \frac{1}{2}$; then [7]

$$|\mathcal{A}_\delta| = \sum_{0 \leq e \leq \delta T} \binom{T}{e} \delta^e (1-\delta)^{T-e} \leq 2^{Th(\delta)}. \quad (1.98)$$

This is a consequence of

$$\begin{aligned} 1 &= (\delta + (1-\delta))^T \\ &\stackrel{(a)}{\geq} \sum_{0 \leq e \leq \delta T} \binom{T}{e} \delta^e (1-\delta)^{T-e} = \sum_{0 \leq e \leq \delta T} \binom{T}{e} (1-\delta)^T \left(\frac{\delta}{1-\delta}\right)^e \\ &\stackrel{(b)}{\geq} \sum_{0 \leq e \leq \delta T} \binom{T}{e} (1-\delta)^T \left(\frac{\delta}{1-\delta}\right)^{\delta T} = \delta^{\delta T} (1-\delta)^{(1-\delta)T} \sum_{0 \leq e \leq \delta T} \binom{T}{e}. \end{aligned} \quad (1.99)$$

Here (a) follows from Newton's binomial, leaving out some terms, and (b) follows from the fact that $\delta/(1-\delta) \leq 1$ and $e \leq \delta T$. From (1.98) we now easily obtain (1.96)

$$R = \frac{\lceil \log_2 |\mathcal{A}_\delta| \rceil}{T} \leq \frac{\log_2 |\mathcal{A}_\delta| + 1}{T} = h(\delta) + \frac{1}{T}. \quad (1.100)$$

(B) Next we derive an upper bound for the probability of occurrence of a sequence outside a Hamming ball. Let $0 < \theta < \delta \leq 1$; then

$$\sum_{\delta T \leq e \leq T} \binom{T}{e} \theta^e (1-\theta)^{T-e} \leq 2^{-Td(\delta||\theta)}. \quad (1.101)$$

This can be shown by first considering the case where $\delta < 1$. Then

$$\begin{aligned} 1 &= (\delta + (1-\delta))^T \stackrel{(a)}{\geq} \sum_{\delta T \leq e \leq T} \binom{T}{e} \delta^e (1-\delta)^{T-e} \\ &\stackrel{(b)}{=} \sum_{\delta T \leq e \leq T} \binom{T}{e} \left(\frac{1-\delta}{1-\theta} \right)^T \left(\frac{\delta(1-\theta)}{\theta(1-\delta)} \right)^e \theta^e (1-\theta)^{T-e} \\ &\stackrel{(c)}{\geq} \sum_{\delta T \leq e \leq T} \binom{T}{e} \left(\frac{1-\delta}{1-\theta} \right)^T \left(\frac{\delta(1-\theta)}{\theta(1-\delta)} \right)^{\delta T} \theta^e (1-\theta)^{T-e} \\ &= \left(\frac{\delta}{\theta} \right)^{\delta T} \left(\frac{1-\delta}{1-\theta} \right)^{(1-\delta)T} \sum_{\delta T \leq e \leq T} \binom{T}{e} \theta^e (1-\theta)^{T-e}. \end{aligned} \quad (1.102)$$

Here (a) follows again from Newton's binomial, (b) is just rewriting, and (c) follows from $\delta(1-\theta) > \theta(1-\delta)$ and $e \geq \delta T$. Finally we consider the case where $\delta = 1$. Then $d(\delta||\theta) = -\log_2(\theta)$. From (1.101) we easily obtain (1.97) since

$$P_\epsilon = \Pr\{X^T \notin \mathcal{A}_\delta\} = \sum_{\delta T \leq e \leq T} \binom{T}{e} \theta^e (1-\theta)^{T-e} \leq 2^{-Td(\delta||\theta)}. \quad (1.103)$$

This finishes the proof of Theorem 1.13. ■

1.5.8 The Fixed-Length Coding Theorem

Theorem 1.14. *For a binary memoryless source with parameter $\frac{1}{2} > \theta > 0$ there exist source codes with rates $1 \geq R > h(\theta)$ that achieve*

$$\lim_{T \rightarrow \infty} P_\epsilon = 0. \quad (1.104)$$

Moreover for $\theta = 0$ or $\theta = \frac{1}{2}$ there exist codes with rates $R = h(\theta)$ and error probability $P_\epsilon = 0$. Therefore we say that the entropy $h(\theta)$ is achievable.

Proof. The proof of this theorem is simple. Just choose $\delta \leq \frac{1}{2}$ such that $h(\delta) = R$. By the monotonicity of the binary entropy function $\frac{1}{2} \geq \delta > \theta > 0$. Therefore $d(\delta||\theta) > 0$. The result is trivial for $\theta = 0$ or $\theta = \frac{1}{2}$. ■

So far we have looked only at binary memoryless sources with $0 \leq \theta \leq \frac{1}{2}$. Note, however, that for $\theta > \frac{1}{2}$ we can interchange zeros and ones in the source sequences. Then $h(1-\theta)$ is seen to

be achievable. However, by the symmetry of the binary entropy function around $\theta = \frac{1}{2}$ we know that $h(1 - \theta) = h(\theta)$. Therefore for binary memoryless sources with parameter $0 \leq \theta \leq 1$ the entropy $h(\theta)$ is achievable.

1.5.9 Converse and Conclusion

We have shown that entropy can be achieved. However, it is possible for even better codes to exist. Since we have chosen code sequences for only the most probable source sequences our coding methods are optimal. Nevertheless this does not yet imply that we cannot achieve compression rates below entropy with vanishing error probabilities.

So, consider our optimal coding scheme again, but now with $\delta < \theta$. So, with (1.100) we may conclude that for T large enough the code rate will be below the source entropy $h(\theta)$. However, the error probability of this scheme can be bounded in a way similar to (1.103), only now we consider $P_c = 1 - P_\epsilon$. Again we can show that

$$P_c = \Pr\{X^T \in \mathcal{A}_\delta\} = \sum_{0 \leq e \leq \delta T} \binom{T}{e} \theta^e (1 - \theta)^{T-e} \leq 2^{-Td(\delta||\theta)}. \quad (1.105)$$

So, the probability for correct decoding is bounded from below and arbitrarily small error probabilities cannot be achieved.

We have looked here only at the binary case. But the results of the previous sections can be generalized to memoryless sources with non-binary alphabets as well.

In this chapter we have discussed aspects of information theory relevant to lossless compression. There are a number of books available that deal with the subject of information. Two highly popular books are those by Cover and Thomas [2] and Gallager [3].

1.6 REFERENCES

1. Abramson, N., 1963. *Information Theory and Coding*, pp. 61–62. McGraw-Hill, New York.
2. Cover, T. M., and J. A. Thomas, 1991. *Elements of Information Theory*. Wiley, New York.
3. Gallager, R. G., 1968. *Information Theory and Reliable Communication*. Wiley, New York.
4. Gray, R. M., and L. D. Davisson, 1986. *Random Processes: A Mathematical Approach for Engineers*. Prentice-Hall, Englewood Cliffs, NJ.
5. Hartley, R. V. L., 1928. The transmission of information. *Bell Systems Technical Journal*, Vol. 17, pp. 535–564, July 1928.
6. Jelinek, F., 1968. *Probabilistic Information Theory*, pp. 476–489, McGraw-Hill, New York.
7. van Lint, J. H., 1992. *Introduction to Coding Theory*. Springer-Verlag, Berlin/New York.
8. Pasco, R., 1976. *Source Coding Algorithms for Fast Data Compression*, Ph.D. thesis. Stanford University, Palo Alto, CA.
9. Rissanen, J., 1976. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, Vol. 20, p. 198.
10. Shannon, C. E., 1948. A mathematical theory of communication. *Bell Systems Technical Journal*, Vol. 27, pp. 379–423, July 1948. Reprinted in *Key Papers in the Development of Information Theory* (D. Slepian, Ed.), pp. 5–18, IEEE Press, New York, 1974.
11. Shields, P. C., 1987. The ergodic and entropy theorems revisited. *IEEE Transactions on Information Theory*, Vol. IT-33, pp. 263–266, March 1987.

Complexity Measures

STEPHEN R. TATE

OVERVIEW

In this chapter, an alternative measure of information, or string complexity, is introduced. The “Kolmogorov complexity” of a string is basically the length of the shortest program that can compute the string, which turns out to be a very fundamental and important way of defining information. We first discuss some problems with traditional information theory, as introduced in the previous chapter, and then give the basic definitions and properties of Kolmogorov complexity and discuss why this theory solves some of the problems inherent in traditional information theory. The chapter next explores the fundamental properties of this complexity measure, including the computational complexity of determining Kolmogorov complexity, showing that unfortunately the Kolmogorov complexity is uncomputable. Finally, it is shown that Kolmogorov complexity and traditional information theory agree in a very strong way in situations where both may be applied.

2.1 INTRODUCTION

A great deal of the progress in data compression owes its existence to the foundation and insights provided by Claude Shannon in his development of information theory, the basics of which were described in the previous chapter. While the very real and practical impact of Shannon’s ideas is obvious, there are some fundamental questions and problems that this theory does not adequately address. The material presented in this chapter shows that, in a very strong sense, there is an ultimate answer to questions about compressibility of data. To distinguish between the two fundamentally different approaches, in this chapter classical information theory will be referred to as “Shannon Information Theory,” and the approaches presented here will be referred

to as “Algorithmic Information Theory” or, by the more commonly used term, “Kolmogorov complexity.”

It should be pointed out here that the motivation for Algorithmic Information Theory is a solid understanding of what information is, and this theory can be used for reasoning about information and incompressibility. However, the practical uses (meaning uses that lead to implementation by programs) of this theory for data compression are somewhat limited. In the practical arena, Shannon Information Theory wins by virtue of its ability to be implemented, despite the less-than-perfect theory of information.

We continue this chapter with an overview of the shortcomings of Shannon Information Theory, followed by an overview of Algorithmic Information Theory and how it addresses these shortcomings, then continue with a more formal treatment of the subject, and finally end with some historical notes on the development of this theory.

Some topics in this chapter assume familiarity with basic computer science concepts and terminology, such as asymptotic notation and computability issues. Of course, this chapter just barely scratches the surface of Algorithmic Information Theory, with a concentration on those issues that relate to data compression. There are an astounding number of applications which are not obvious at all initially. For example, Algorithmic Information Theory can be used to provide an estimate of how many prime numbers are in a particular range, even though at first this question seems to have little to do with compression or even information theory in a standard sense. A book by Li and Vitányi [10] contains a thorough treatment of Algorithmic Information Theory for those who wish to pursue this area more fully. In this chapter, we use notation that mostly agrees with that in the Li and Vitányi book, although this chapter was written with a conscious effort to use terminology that is less formal and more familiar for non-experts when possible (e.g., we use “computable” instead of “recursive”).

2.1.1 An Aside on Computability

While non-computability is a well-known topic for people who have studied computer science, some results come as a surprise to those who have not studied the area. It is not surprising at all that there are functions which are so complex that it takes a long time to compute them on current computer hardware. For example, factoring the product of two large prime numbers (without knowing the primes, of course!) seems to be extremely difficult, and in fact much of the security of modern on-line commerce depends on this function being very difficult to compute. However, this is very different from saying that the function is non-computable—in fact, factoring *is* computable because there is a very simple algorithm to do so: Simply try dividing all possible divisors into the number in question. The fact that this algorithm, run on numbers used for cryptography today, would take longer than the lifetime of the universe to complete is very important for the efficiency of the procedure, but completely irrelevant for whether the function is *computable*, where only the existence of the algorithm is important.

If computers were suddenly to become 10^{30} times faster, then all of a sudden we could factor the numbers in question, and a lot of “security” would become not so secure. This is in stark contrast to the notion of non-computability, which means that no algorithm exists at all. Computers could become 10^{30} times faster, or even 10^{1000} times faster, and it would have no effect—non-computable is non-computable, regardless of time or speed or memory available. This fact is surprising for many people who seem to have an initial intuition that everything is computable, given enough time.

The fact that there are some functions that simply cannot be expressed by an algorithm is unusual, but has parallels in other areas of mathematics. For example, Gödel’s incompleteness theorem says that in any logical system (defined by a basic set of axioms), there are statements

that are true, but cannot be proved true—this is in fact very closely related to non-computability results, where the algorithmic language corresponds to the logical system, and the true statements that can be defined but not proved correspond to functions which can be defined but not computed.

In reading this chapter, keep in mind that when we refer to non-computable functions, we are not talking about simply a lack of power or speed of current machines, nor are we talking about a function which we simply have not been clever enough to invent an algorithm for. We are talking about a fundamental incomputability that exists on all conceivable computers.

2.2 CONCERN WITH SHANNON INFORMATION THEORY

Shannon Information Theory provides some remarkable insight into the information content of data sources, but fails to satisfactorily answer some natural questions. We outline a few issues with Shannon Information Theory below, all of which will be addressed in a satisfactory way by Algorithmic Information Theory.

2.2.1 Strings versus Sources

The most fundamental question in information theory is, given a string x , what is the information content of that string? Shannon Information Theory does not have the ability to address this question, as it must look at this in terms of a probabilistic data source. In particular, if we are considering only one possible string, you might say the probability of this string is 1, which leads to the result that the entropy is zero and the string contains no information. This contradicts common sense and argues for a theory that can describe the information content of individual strings.

2.2.2 Complex Non-random Sequences

The digits of π form an infinite sequence with many properties in common with truly random sequences, such as equal frequency of digits and strings of digits. By trying to apply Shannon Information Theory and forcing a probabilistic source on top of the stream of digits, the result is a source that offers no compression due to the apparent randomness of the digits. This again contradicts what we know to be true, since the digits of π are *extremely* compressible. In fact n digits can be represented in $O(\log n)$ space, since all we really need is a program that knows how to compute the digits of π and an indication of the length of the string that we need to generate.

This argument extends to a different problem which turns out to be related to the same issue. Consider the output of a good pseudo-random number generator (by “good” we mean one where the output can pass all known statistical tests for randomness—cryptographic pseudo-random number generators certainly meet this requirement). Since the output appears random, Shannon Information Theory offers no help here, but again the data stream can be very highly compressed. The pseudo-random stream can be completely described by simply giving the length of the output and the seed required to start the pseudo-random number generator.

2.2.3 Structured Random Strings

Consider a random binary data source in which each bit is uniformly and independently chosen (so each bit has probability $\frac{1}{2}$ of being 0 and probability $\frac{1}{2}$ of being 1). Shannon Information Theory tells us that data from such a source cannot be compressed on average and, in fact,

compression approaches that are based on Shannon Information Theory (such as Huffman coding or arithmetic coding) will code all data from this source using the same size code and will achieve no compression on any string. But is it accurate to say that no data from this source are compressible?

Consider two strings, “00000000000000000000” and “0101001101110011100.” Most people would say the second string is “more random” and “less compressible” even though there is absolutely no justification for these statements in light of the data source, as both strings have exactly the same probability of occurrence (they both occur with probability 2^{-20}). By Shannon’s definitions, both strings have the same information content, although most people would disagree with this statement.

Each of the problems outlined above has a distinct flavor, but there is one common underlying theme: An information measure should reflect the difficulty of constructing or computing the data, not some probabilistic model that we try to fit to the data. With that in mind, we consider an information measure that addresses the descriptive complexity of an algorithm for computing the data and refer to this as Algorithmic Information Theory. This approach not only satisfactorily addresses the concerns we outlined above, but also matches up quite nicely with Shannon Information Theory in situations where that theory is applicable.

To this point, we have used the term Algorithmic Information Theory to emphasize that this is, in fact, a theory of information in the same sense as Shannon’s theory. However, a more common term for this theory is Kolmogorov complexity, named after the Russian mathematician who was one of the originators of the theory. As will be described in the last section of this chapter, on the history of Kolmogorov complexity, this term is not without controversy since similar ideas were independently discovered at roughly the same time by Solomonoff and by Chaitin. However, to match common usage, in the remainder of this chapter we will use the term Kolmogorov complexity, as this has emerged as the most commonly used name.

2.3 KOLMOGOROV COMPLEXITY

The basic idea of Kolmogorov complexity is to associate information content with the difficulty of describing data, whether or not we can model that data as coming from a probabilistic source as in Shannon Information Theory. The notion of “describing data” can be made more concrete by allowing any *process* which can create the data in question, and a process is simply a computational procedure or algorithm. This then is the rough definition of Kolmogorov complexity: The Kolmogorov complexity of a string x is the length of the shortest program that generates x .

There are two important points to understand from this rough definition. First, only the size of a program is mentioned, but people familiar with compression generally think in terms of two items: a program (the decompressor) and its input (the compressed data). However, this is not really such an odd restriction. Since we are considering only a single instance (i.e., one output string), the input to a decompression program is fixed and can, in fact, be included in the program itself. Second, notice that every string does have a finite program that generates it. While we delve into the question of “what is a program” below, every reasonable programming language could produce any string with a single long “print statement,” for example, “Print (010001110110001010100010010101).”

While many different types of computing machines are available, the notion of what is computable is extremely robust across all the options, from real physical machines to mathematical models (including more esoteric models such as quantum computers) and even to our understanding of how the human brain processes information. In fact, the “Church–Turing thesis” states that the intuitive notion of computability agrees exactly with what can be computed by current

formal universal models of computation, such as a Turing machine or a random access machine [6]. While the Church–Turing thesis has not been proved (and *cannot* be proved unless something more precise than an “intuitive notion” is involved), it is generally accepted to be true. So in considering a specific model of computation, such as a Turing machine, we do not restrict the power of what we can compute.

Associating the smallest description of data with a metric for the validity or information content of the data is not unique to Kolmogorov complexity. Other areas, including other branches of mathematics and philosophy, also express similar ideas. For instance, in statistics there is a theory known as the “minimum description length principle,” or just “MDL,” which states that if you are given a collection of theories that describe some data, the most appropriate one is the one that minimizes the sum of the sizes of the theory description and the encoding of the data under that theory. As such theories must be describable, and hence computable, this is clearly just a restatement of Kolmogorov complexity, and in fact Kolmogorov complexity has been put to great use in the area of statistics and inductive reasoning. Furthermore, these concepts correspond very nicely to the philosophical statement known as “Occam’s Razor”: given competing explanations for an event, the simplest one is usually right.

2.3.1 Basic Definitions

As described above, the difficulty of describing data can be identified with the difficulty of describing an algorithm that produces the data. Note that we are not concerned (at least for now) with the computational complexity of this process, but rather only with the difficulty of describing the algorithm. So how are algorithms expressed, and how can we measure the complexity of an algorithm description?

The choice of measurement is a standard one: We represent data with binary strings, and the complexity of such a string is simply its length. In this chapter, we use the notation $|x|$ to denote the length of a binary string x . While the choice of a binary coding alphabet is straightforward, the semantics of such a string are unclear—in other words, given a binary string x , what does it represent? There are many possible choices: perhaps x is a binary encoding of a Turing machine, or x is the ASCII representation of a LISP program, or x is a binary executable for some specific machine architecture and OS.

We will use the term “descriptive language” to refer to such a language,¹ where the only condition is that it be possible to actually execute programs from this language. In particular, let p be a program written in such a language, and y be an input to p , and let ϕ be the function that maps the pair $\langle p, y \rangle$ to the output when program p is run with input y . Then the function ϕ must be computable (or more precisely, ϕ must be a partial recursive function). As the language and the function that simulates it (ϕ) are really the same, these terms can be used interchangeably.

Definition 2.3.1. *The Kolmogorov complexity of a string x with respect to a descriptive language L is the shortest program in L that produces x as output. We denote this complexity by $C_L(x)$.*

An example should clearly demonstrate that selection of the descriptive language can make a very real difference. Consider the following base 10 number: 51090942171709440000. This number is exactly $21!$ (21 factorial), but this number requires over 65 bits to represent in binary.

¹ Note that we use the term “language” here in the sense of programming languages, which have both syntax and semantics. This should not be confused with the formal languages sense of a “language” which defines just a set of strings with no particular semantics.

Although you could write a program in C to compute this value, numbers this large cannot be directly represented, at least on current machines. So, in addition to the factorial program, you would have to write routines to do arithmetic on “big numbers.” As a result, the size of this program would be substantially larger than just printing out the number. On the other hand, if our descriptive language is the Mathematica programming language, then this value can be produced by the three-character program “`21!`,” since Mathematica has intrinsic support for both the factorial function and for working with large numbers. In other words, for this string $C_{\text{Mathematica}}(x)$ is much smaller than $C_C(x)$.

Although the choice of representation does indeed make a difference, as just shown, the difference turns out not to be terribly great for an important and common class of languages called “universal languages.” In fact, by definition, any universal language (sometimes called a “Turing complete language”) can simulate any other universal language, and the size of such a simulation program is a constant that depends only on the two languages and not on the program that is being simulated. For example, we can certainly simulate full Mathematica functionality with a constant-size C program (in fact, Mathematica may very well be written in C). This leads us to what is called the “Invariance Theorem.”

Theorem 2.3.1. *If f and g are two universal languages, then for all strings x , $|C_f(x) - C_g(x)| \leq c_{f,g}$, where $c_{f,g}$ is a constant that depends only on f and g .*

At times (such as in the proof of Theorem 2.4.1 later in this chapter), it is convenient to consider arbitrary languages, even if they are not universal. In such a case we can still upper-bound the power of such a language, but since we do not require that all things are computable we cannot lower-bound the power of this language. Thus, the following theorem is a slight variation of the Invariance Theorem.

Theorem 2.3.2. *If f is a universal language, and g is any language defined by a partial recursive function g , then for all strings x , $C_f(x) \leq C_g(x) + c_{f,g}$, where $c_{f,g}$ is a constant that depends only on f and g .*

Theorem 2.3.1 says that as long as the language is universal, it does not matter what it is, up to an additive constant. The example given previously (representing $21!$) showed an extreme difference, but this was only because the data being represented were so small. For large strings, the additive constant difference in complexities becomes insignificant. This independence of language is what makes Kolmogorov complexity have a fundamental meaning that transcends issues of whim such as choice of language. Li and Vitányi [10] stress this point succinctly by noting that “The remarkable usefulness and inherent rightness of the theory of Kolmogorov complexity stems from this independence of the description method.”

Since the choice of language does not make a significant difference (as long as it is universal), we will remove the subscript from the complexity notation and simply refer to $C(x)$, understanding that results are significant only up to an additive constant value. There is in fact an area of study known as “concrete Kolmogorov complexity” in which a specific language is chosen (such as an encoding of a Turing machine with a specific, preferably small, number of states, symbols, and tapes), and then the actual constants can be estimated and evaluated.

2.3.2 Incompressibility

It should be clear that *any* lossless compression technique can be viewed in the light of Kolmogorov complexity, as just described, since we must have a program that can reproduce the original data. Unlike Shannon Information Theory, we do not have to refer to a sometimes artificially created

probabilistic model of the input, but rather can handle anything which can be computed by any machine, and in any language. This universality makes Kolmogorov complexity an extremely powerful theory. So, what can Kolmogorov complexity tell us about compression?

We start with a definition that gives terminology for discussing incompressible strings. Recall that because of the default program of simply printing out the desired string, for any string x we can bound $C(x) \leq |x| + c$, where the constant c simply reflects the size of the “print” program. This is an upper bound, so how do we refer to a lower bound?

Definition 2.3.2. A string x is c -incompressible if $C(x) \geq |x| - c$.

Intuitively, a c -incompressible string is one that cannot be generated by *any* program significantly shorter than the explicit printing program. The c -incompressible strings are then those that basically cannot be compressed, using any program at all, irrespective of the computational time allowed for the program. To examine how prevalent c -incompressible strings are, we take a brief detour through something called the “counting argument,” which is really just an application of the well-known pigeonhole principle.

Theorem 2.3.3. Let $S \subseteq \{0, 1\}^{\leq b}$ be a subset of binary strings of length at most b , and let $f : S \rightarrow \{0, 1\}^*$ be any function mapping from S to arbitrary binary strings. The image of S under f (sometimes called the “range” of f) contains at most $2^{b+1} - 1$ distinct values.

Proof. Since there are no restrictions on how f is defined, the number of distinct values produced is limited only by the size of the domain S . Because there are exactly 2^k binary strings of length k , the number of binary strings of length b or less is exactly

$$|\{0, 1\}^{\leq b}| = \sum_{k=0}^b 2^k = 2^{b+1} - 1.$$

Since S is a subset of $\{0, 1\}^{\leq b}$, this upper-bounds the size of S . ■

As a direct application of this theorem, consider a fixed lossless compression algorithm $A = (f_c, f_d)$, where f_c is the compression function and f_d is the decompression function. Consider the following question: How many strings of length n can be compressed by A with a savings of at least 1 byte (8 bits)? The decompression function f_d must be able to reconstruct each original binary string given its compressed representation, and the compressed forms of those that are compressed by at least 1 byte form a subset of $\{0, 1\}^{\leq b}$ with $b = n - 8$. Theorem 2.3.3 now tells us that f_d can have at most $2^{b+1} - 1 = 2^{n-7} - 1$ possible distinct outputs, so this limits the number of files that can be compressed with 1 byte of savings. Because there are 2^n possible input files, there must be at least $2^n - 2^{n-7} + 1 > \frac{127}{128}2^n$ files that *cannot* be compressed by a byte. Without more knowledge of how A works, we cannot determine which strings can or cannot be compressed. But because there are 2^n possible inputs of length n , a randomly selected file of length n will be incompressible (by a byte or more) with probability $> \frac{127}{128} > 0.99$.

The above argument applies to a single compression system A and can be loosely summarized by saying “every compression algorithm will fail to achieve compression on a significant fraction of possible input files.” But with Kolmogorov complexity, each input is allowed to have its own, custom-built decompression program, so can we say anything as strong? It turns out that the argument for Kolmogorov complexity is really the same and simply requires a slightly different perspective.

When we are computing the Kolmogorov complexity of a string, we always do so with respect to a fixed, specific descriptive language, say L . Because it must be possible to interpret any such language using any other universal language, we can consider the function f of Theorem 2.3.3

to be simply an “interpreter” for language L and apply the theorem to lower-bound the number of incompressible strings (even with custom-built decompression programs!). The precise result is stated in the following theorem.

Theorem 2.3.4. *Over $1 - 2^{-c}$ of all strings of length n are c -incompressible.*

Consider once again strings which allow compression by at least 1 byte. Using $c = 7$ in Theorem 2.3.4 shows that at least a fraction $1 - 2^{-7} = \frac{127}{128}$ of strings of length n is not compressible by even a single byte (note that $c = 7$ since any savings strictly less than 8 bits is not counted)! This is the same fraction we obtained in the earlier argument, but notice how much more powerful our setting has become. The earlier argument says that given a single, fixed compression system, this fraction of files is not compressible in this system. On the other hand, this latest argument says that this fraction ($\frac{127}{128}$) of files is not compressible by *any* compression program, even allowing compression programs to be tailored specifically for the data in question!

Trying an additional value of c is enlightening. With $c = 55$, the c -incompressible strings are those that cannot be compressed, by any program, with a savings of 7 or more bytes. Note that for files 70,000 bits (8750 bytes) or larger, the 55-incompressible strings are those that cannot be compressed by even 0.01%. Theorem 2.3.4 states that at least $1 - 2^{-55}$ strings are 55-incompressible, or, in other words, the probability of a randomly selected file being compressible by 7 or more bytes is less than 2^{-55} . To appreciate how small this number is, consider that Bruce Schneier quotes 2^{-55} as the probability that a person would win a major state lottery and get hit by lightning on the exact same day (Table 1.1 of [11]).

2.3.3 Prefix-free Encoding

A set of binary strings is “prefix-free” if no string in the set is a prefix of any other string in the set. Prefix-free sets, often referred to as “prefix codes,” are useful because of their self-delimiting nature: An algorithm reading a binary string can determine when it reaches the end of a codeword, without having to look beyond the end of the codeword. Because of this property, codewords can be concatenated to form longer strings of codewords, which can then be easily parsed into the individual components. In Shannon Information Theory, only prefix codes are typically considered.

In Kolmogorov complexity, as defined and explained in the previous section, the string encodings are not required to be prefix-free. The limits of these strings are typically determined by some “out-of-band” information. For example, a string encoded by a program as required by Kolmogorov complexity might be delimited by placing it on a Turing machine tape surrounded by blank cells.

Unfortunately, this leads to some minor difficulties in the general theory of Kolmogorov complexity. For example, as defined previously, Kolmogorov complexity is not “subadditive.” In other words, it is intuitively appealing that the complexity of a pair of strings, x and y , is no more than the sum of the complexities of the individual strings (so $C(x, y) \leq C(x) + C(y)$). However, this is unfortunately not true with the previous definitions.

To correct these problems, Kolmogorov complexity is often studied in a prefix-free form, called “Algorithmic Prefix Complexity.” In this case, the language being used to define programs must ensure that the set of all programs is prefix-free. In fact, this is not so unusual. Typical binary executable formats on real machines have this property (since the length of the executable is in the header), as do some high-level programming languages (such as Pascal, where the end of the program is always marked by the string “end.”).

We use the notation $K(x)$ to refer to the algorithmic prefix complexity of a string x . Clearly, adding the restriction that the set of possible programs be prefix-free cannot *decrease* the

complexity of a string, but how much can it increase? Fortunately, since this is a relatively minor change in the notion of complexity, the difference is not too great. Any set of programs can be converted to a self-delimited, prefix-free set in the following manner. Let p be the shortest program for computing x (so $C(x) = |p|$). First figure out how many bits are in the binary representation of $|p|$, the length of p . Then write out that many zeros, followed by $|p|$ in binary (note that it must start with a 1 bit), followed finally by p itself. This is clearly a self-delimiting description, so the set of all such descriptions is prefix-free. Furthermore, since the number of bits in $|p|$ is at most $\lceil \log_2 |p| \rceil$, we have $K(x) \leq C(x) + 2\lceil \log_2 C(x) \rceil$.

In fact, we can make this a little stronger still by repeating this self-delimiting process. In other words, to encode $|p|$ we really do not need to write it in binary, but rather need only to write the shortest self-delimiting program for generating $|p|$ which, by the above argument, is at most $C(|p|) + 2\lceil \log_2 C(|p|) \rceil = C(C(x)) + 2\lceil \log_2 C(C(x)) \rceil$, and so

$$K(x) \leq C(x) + C(C(x)) + 2\lceil \log_2 C(C(x)) \rceil.$$

This can, in fact, be repeated again to get an even tighter bound. The following theorem, attributed to Solovay [14] in [10] and given here without proof, makes the connection between the “ K -complexity” and “ C -complexity” quite explicit.

Theorem 2.3.5. *For any string x ,*

$$K(x) = C(x) + C(C(x)) + O(C(C(C(x)))),$$

and

$$C(x) = K(x) - K(K(x)) - O(K(K(K(x)))).$$

2.3.3.1 Subtle Problems with Non-prefix Codes

Issues with prefix-free versus non-prefix-free codes can be very subtle and can cause a great deal of confusing and misleading statements. Several people have mistakenly claimed to have techniques for compression of arbitrary data by not understanding this concept fully. Typically, these claims go along the following lines: I can take all strings of length n and represent them with codes whose average length is less than n , so simply repeat this process with an appropriate block size and compression is achieved! In fact, the first part of the claim is true: If the input consists only of strings of length n , then these strings can be encoded with a variable-length code whose average length is approximately $n - 2$. However, the input set is in fact prefix-free (any set of fixed-length strings is prefix-free), and the second set is not. By encoding from a prefix-free set to a non-prefix-free set, there is additional information “hidden” in the output that corresponds roughly to the length of the variable-length output. Although this is simply a curiosity for single blocks, it turns into a problem when this process is repeated (as in the second part of the false “compress anything” claim). Without some sort of external information present, it is impossible to extract these variable-length encoded blocks of data, so the reverse transformation cannot be performed.

Note that an honest accounting for code lengths would require that fixed-size blocks be coded with a prefix-free set of codewords, even if variable length is allowed. If this one restriction is included, then no variable-length code can have average codeword length less than n (the proof uses the Kraft inequality), thus making the first part of the compress anything claim impossible.

The importance of prefix codes to Kolmogorov complexity is highlighted by a recent turn of events. Due to the regular claims of new revolutionary compression algorithms that work for arbitrary data, several “challenges” exist on the comp.compression Usenet newsgroup. One such challenge is set up so that a challenger can send \$100 to the challenge author, who will

send a data file of any requested length to the challenger. If the challenger can produce a program and data file whose combined length is less than that of the original data and yet manages to reproduce that data, then the challenge author will send \$5000 to the challenger. This challenge is motivated exactly by Kolmogorov complexity and the ideas presented in this chapter. Despite the allowance of separate program and data, which allows a little bit of “cheating” on the Kolmogorov complexity, we know by Theorem 2.3.4 that if the challenge data are generated randomly, then there is an extremely low probability that the challenger will succeed.

This challenge was originally designed to dissuade people from making “arbitrary compression” claims without clearly thinking them through, but the challenge was taken up in April 2001 with some interesting results. In a series of e-mails with the challenge author, the challenger managed to slip in a change in the rules that allowed for multiple data files rather than a single file. The “decompressor,” subsequently provided by the challenger, then simply concatenated the multiple data files together with a fixed separator byte value. Since the separation of any two pieces saves a byte by not being prefix-free, adding up this savings over several hundred “pieces” results in saving several hundred bytes. This is enough to offset the size of the program to reassemble the pieces. The result is that the apparent total data size (what you get if you simply add up the sizes of all files involved) has in fact decreased. This worked entirely because a very subtle problem in the original challenge statement (allowing non-prefix-free representations) was exaggerated by repeating that problem many times over.²

2.4 COMPUTATIONAL ISSUES OF KOLMOGOROV COMPLEXITY

Since $C(x)$ is in a very strong sense the ultimate answer to the information content of the string x , the next natural question is “how can we measure $C(x)$?” The unfortunate answer to this question is that we cannot. There is no method or algorithm for computing $C(x)$ on any infinite set of strings that is correct over its entire domain. In fact, while upper bounds on $C(x)$ are possible in some sense, $C(x)$ cannot be lower-bounded in a reasonable fashion by any algorithm! In this section, these concepts are explained and expanded. We present a sequence of results that are increasingly powerful but require an increasing amount of knowledge of recursion theory (or computability theory).

In the proofs that follow, a specific enumeration of binary strings (which represent programs) is required. Specifically, for two strings x and y , we order $x < y$ if $|x| < |y|$ or if the lengths are the same and the binary number represented by the string x is smaller than the number corresponding to y . Clearly this enumeration can be listed out in order: Simply have a binary counter for strings of a particular length and place that inside a loop over all lengths in increasing order.

The following basic theorem says that no algorithm can correctly compute $C(x)$. The proof is very basic and does not require any knowledge of recursion theory. It is hopefully understandable to anyone who has followed the basic definitions in this chapter.

Theorem 2.4.1. *There is no algorithm that correctly computes $C(x)$ for all strings.*

Proof. Assume for the sake of contradiction that $C(x)$ is a computable function. Consider the following function: $f(m) = \min\{x \mid C(x) \geq m\}$, where the “min” is with respect to the binary string enumeration described above. Put into words, $f(m)$ is a function which maps m to the first string whose Kolmogorov complexity is at least m . This is well defined because for a given

² Note that in addition to the savings allowed by exploiting the non-prefix-free “loophole,” there was in fact some information encoded in the *names* of the data files as well, since it was the names that allowed the pieces to be put back together in the proper order.

value of m , only a finite number of strings can have Kolmogorov complexity smaller than m . The important thing to notice is that if $C(x)$ is computable, then $f(m)$ can be computed as well, simply by going through an enumeration of strings x and testing $C(x)$ for each string in order.

Since f is a computable function (a “total recursive function” in recursion theory terms), we can use it as a descriptive language for Kolmogorov complexity and consider $C_f(f(m))$. Clearly, the input to f which produces $f(m)$ can simply be the integer m , which can be encoded with $\lceil \log_2 m \rceil$ bits, and hence $C_f(f(m)) \leq \lceil \log_2 m \rceil$. Furthermore, by the Invariance Theorem (Theorem 2.3.2) we know that $C(f(m)) \leq C_f(f(m)) + c$, where c is a constant that does not depend on m . Hence, $C(f(m)) \leq \lceil \log_2 m \rceil + c$. However, by the definition of $f(m)$ we know that $C(f(m)) \geq m$, which combined with the last observation implies that $m \leq \lceil \log_2 m \rceil + c$. Regardless of the specific value of c , it is always possible to find an m large enough to violate this condition, so we have reached a contradiction. Therefore, our original assumption, that $C(x)$ is computable, must be incorrect, which completes the proof of this theorem. ■

Since $C(x)$ cannot be computed by an algorithm that is correct all the time, the next question to ask is “Can $C(x)$ be computed correctly for some infinite set of strings?” The answer to this is also “no.” The precise statement of this result, as well as its proof, requires a knowledge of basic recursion theory, such as the definition of “recursively enumerable” and basic properties of recursively enumerable sets.

Theorem 2.4.2. *Let $g(x)$ be a partial recursive function with infinite domain A . Then $g(x)$ cannot be equal to $C(x)$ on its entire domain.*

Proof. Assume that $g(x) = C(x)$ over its entire domain. Since $g(x)$ is partial recursive, its domain A is recursively enumerable. Since every infinite recursively enumerable set has an infinite (total) recursive subset, let $B \subseteq A$ be an infinite recursive set. Consider the function $f(m) = \min\{x \in B \mid C(x) \geq m\}$. Since B is total recursive, we can test for membership in B , and for all $x \in B$ we can use the function $g(x)$ to compute $C(x)$. (Recall that by assumption $g(x) = C(x)$ for all $x \in B$.) This means that $f(m)$ is a total recursive function. Given this function $f(m)$, we can repeat the proof of the preceding theorem in order to reach a contradiction, and so our conclusion is that $g(x)$ cannot agree with $C(x)$ over its entire domain. ■

We take a short break now from examining things that we *cannot* compute to look at something that we can.

2.4.1 Resource-Bounded Kolmogorov Complexity

In some sense, a good deal of computational complexity theory arose by bounding various resources and considering ideas from recursion theory. For example, placing a polynomial time bound on computations, the set of recursive languages naturally corresponds to the class P , and the set of recursively enumerable languages roughly correspond to NP . We get similarly interesting results by placing bounds on the algorithms described in Kolmogorov complexity. While any measurable resource can be limited, in this presentation we consider only limiting the time complexity of the computations.

Definition 2.4.1. *$C_L^t(x)$ is the time-bounded Kolmogorov complexity of x , which is the length of the shortest program p in descriptive language L such that $L(p) = x$ and $L(p)$ runs in time $\leq t$. There is no guarantee that such a program p even exists and, if there is no such program, we define $C_L^t(x) = \infty$.*

While $C(x)$ is not computable, clearly $C_L^t(x)$ is: Simply enumerate all programs from shortest to longest, simulating each one for t steps, and stop once $L(p) = x$. Since we have an upper

bound on the running time, we have a point where we can “move on” to the next possible program, which removes precisely the uncertainty that made the unbounded Kolmogorov complexity uncomputable.

It is worthwhile to stress again that we are interested in what is computable, not what is efficiently computable. The above argument shows that $C'_L(x)$ is computable, but this argument is still not particularly useful for practical applications. The algorithm described above is a valid algorithm, but the time required would be astronomical and so could not be used in practice.

One nice consequence of time-bounded Kolmogorov complexity is that it gives us a way to compute an upper bound for the complexity of a string. Noticing that $C_L(x) = \lim_{t \rightarrow \infty} C_L^t(x)$, we have a simple way of proving the following theorem (Theorem 2.3.3 from [10]), where the function $g(x, t)$ referred to is just $C_L^t(x)$.

Theorem 2.4.3. *There exists a total recursive function $g(x, t)$, monotonically decreasing in t , such that $\lim_{t \rightarrow \infty} g(x, t) = C(x)$.*

2.4.2 Lower-Bounding Kolmogorov Complexity

In the previous section, we saw that we can in some sense upper-bound the Kolmogorov complexity of a string, and so now we turn to lower bounds. Unfortunately, lower bounds are much more difficult. In fact, there does not exist any computable, monotonically non-decreasing and unbounded function which is a lower bound for the Kolmogorov complexity at an infinite number of points.

First consider the Kolmogorov complexity itself. There are a large number of strings with a very small Kolmogorov complexity, and in fact some extremely long strings can be generated by very small programs, so it is natural to ask whether there exists an unbounded monotonically non-decreasing lower bound for the Kolmogorov complexity (note that this would not exist if, for example, there were an infinite set on which the Kolmogorov complexity is bounded by a constant). It turns out that there is such a lower bound.

Consider the function

$$m(x) = \min\{C(y) | y \geq x\}.$$

Clearly $m(x)$ is monotonically non-decreasing and lower-bounds the Kolmogorov complexity (i.e., $m(x) \leq C(x)$ for all x). In fact, $m(x)$ is the greatest monotonically non-decreasing lower bound for the Kolmogorov complexity. The following theorem answers the question from the previous paragraph.

Theorem 2.4.4. *$m(x)$ is unbounded.*

Proof. Assume for the sake of contradiction that $m(x)$ is bounded, so $m(x) \leq b$ for some bound b . Create a sequence of strings by setting $x_1 = \min\{x | C(x) \leq b\}$ and $x_i = \min\{x > x_{i-1} | C(x) \leq b\}$ for $i \geq 2$. Since $m(x)$ is bounded by b , we can always find such an x_i , and so this defines an infinite sequence of distinct strings all with $C(x_i) \leq b$. However, there are only finitely many programs of length $\leq b$, and so only finitely many strings can have Kolmogorov complexity $\leq b$. Thus we have a contradiction, and so $m(x)$ must be unbounded. ■

So $m(x)$ is unbounded, but at what rate does it grow? It turns out that it grows *extremely* slowly—more slowly in fact than *any* computable unbounded function. This is stated more precisely in the following theorem.

Theorem 2.4.5. *There is no computable function $g(x)$ that is unbounded and lower-bounds $m(x)$.*

Proof. This theorem will be proved by contradiction, and the overall structure of the proof is similar to what was done for Theorem 2.4.1. First assume that such a function $g(x)$ exists. We define a function $f(b) = \min\{x | g(x) > b\}$. Since $g(x)$ is computable (i.e., total recursive) and unbounded, $f(b)$ is also computable: A simple algorithm for $f(b)$ simply enumerates strings in increasing order, checking $g(x)$ for each string and stopping once $g(x) > b$. Therefore, to compute $f(b)$ using recursive function f , it is necessary to supply only the value b , which takes at most $\lceil \log_2 b \rceil$ bits, and so $C_f(f(b)) \leq \lceil \log_2 b \rceil$. By the Invariance Theorem this implies that $C(f(b)) \leq \lceil \log_2 b \rceil + c$.

On the other hand, notice that since $m(x)$ is a lower bound for $C(x)$, and $g(x)$ is a lower bound for $m(x)$, we can bound $C(f(b)) \geq m(f(b)) \geq g(f(b)) > b$ (the last inequality comes from the definition of $f(b)$). Combined with the upper bound on $C(f(b))$ found in the previous paragraph, we get $b < \lceil \log_2 b \rceil + c$, which cannot be true for sufficiently large b . This contradiction proves that no such function $g(x)$ can exist, which completes the proof of the theorem. ■

We note here that the preceding theorem can be extended to partial recursive functions without too much difficulty, so the final result for attempts at lower-bounding Kolmogorov complexity is that there exists no partial recursive function that is monotonically non-decreasing, is unbounded, and lower-bounds $C(x)$ at infinitely many points. The proof is a slight modification of the preceding proof and can be found as Theorem 2.3.1 in [10].

2.5 RELATION TO SHANNON INFORMATION THEORY

Now that the basics of Kolmogorov complexity have been explained, it is natural to ask what the relationship is between the two main theories of information: Kolmogorov complexity and Shannon Information Theory. Since these two theories were arrived at from completely different starting points, and taking two very distinct approaches, it is perhaps surprising that in the domain where they can both be applied they are in almost complete agreement. On the other hand, as they both aim to answer the same fundamental questions regarding information content, perhaps this should not be so surprising after all. Since Shannon Information Theory requires codings to be prefix-free, in this section we will use the prefix-free variant of Kolmogorov complexity, as discussed and given the notation of $K(x)$ in Section 2.3.3.

We consider strings generated by a probabilistic source \mathcal{S} , where the set of possible strings is x_1, x_2, \dots, x_m , and the probability of string x_i being generated is p_i .³ The entropy of this source can be written as

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

This is the basically the expected information content (in the Shannon sense) of strings generated by this source, although note that it does not depend in any way on the strings themselves (only the set of probabilities). We can also consider the expected information content, as measured

³ Note that this is slightly different from the traditional definition of a source, because of the inclusion of the “strings.” The strings are not needed, and are in fact irrelevant, for most of Shannon Information Theory, but are vitally important to Kolmogorov complexity. Therefore, to relate these two measures we must include strings in the definition of a source!

by Kolmogorov complexity, of strings from this source. We define the expected Kolmogorov complexity of a source to be

$$\overline{K}(\mathcal{S}) = \sum_{i=1}^m p_i K(x_i).$$

Note that in this measure, the strings themselves (which were ignored in the entropy) are giving the main information content, whereas the probabilities are used only for taking the expected value.

Ideally, we would like to say that $H(\mathcal{S})$ is equal to $\overline{K}(\mathcal{S})$, but unfortunately it is not quite this easy to get a meaningful result. What can be proved without too much difficulty is that these two values are equal up to an additive constant, where the constant depends only on the probability distribution of the source. However, this is a pretty content-free result: Since $H(\mathcal{S})$ itself depends only on the probability distribution, the difference of $\overline{K}(\mathcal{S})$ and $H(\mathcal{S})$ from this argument tells us very little about the actual relationship between these measures. In the following two subsections, two approaches are considered to make the correspondence more meaningful: In the first, we consider a series of sources with increasing entropy, so that functions of the entropy stand out clearly from constants that do not grow with the entropy. In the second approach, we introduce and use conditional complexity measures to reduce the difference between the measures to a small constant that is independent of the probability distribution, showing that these two measures really *are* the same.

2.5.1 Approach 1: An Infinite Sequence of Sources

In this section, let \mathcal{S} be a source as normally considered in Shannon Information Theory: In particular, \mathcal{S} is a stationary, ergodic process (such as an ergodic Markov chain). For the results of this section, we also require that this process have a recursive probability distribution (a recursive probability distribution is simply one in which the probability of any possible output string is computable). Note that models commonly used in Shannon Information Theory fit easily into this setting. For example, if the source is an *iid* (independent, identically distributed) source, then a list of the probabilities of the individual output alphabet symbols describe the source, and string probabilities are obtained by simply multiplying together the individual symbol probabilities. More advanced models, such as Markov chains, also easily work here, but with a slightly more complex method for computing string probabilities.

Let \mathcal{S}_n denote the source restricted to output strings of length n , where probabilities are conditioned on this length. We use $x_{1,n}, x_{2,n}, \dots, x_{m_n,n}$ to denote the possible outputs from this source and $p_{1,n}, p_{2,n}, \dots, p_{m_n,n}$ to denote the probabilities. Remember that these are conditioned on the length of the output, so if A denotes the set of all length n strings, and $P(x)$ denotes the probability of a string x in the original source \mathcal{S} , and $p(A) = \sum_{x \in A} P(x)$, then $p_{i,n} = (P(x_{i,n}))/(P(A))$.

Theorem 2.5.1. *With the above definitions, $0 \leq \overline{K}(\mathcal{S}_n) - H(\mathcal{S}_n) \leq K(n) + c$, where c is a constant that depends only on \mathcal{S} (and in particular, not on n).*

Proof. Lower-bounding $\overline{K}(\mathcal{S}_n)$ turns out to be easy. While Kolmogorov complexity gives much lower complexities for some strings than would be expected from Shannon Information Theory, this cannot allow Kolmogorov complexity (at least in the prefix-free variant) to beat the entropy bound on average! In particular, the minimum-length programs considered in algorithmic prefix complexity form a prefix-free set of codewords, and Shannon's noiseless source-coding theorem states that no uniquely decipherable code (such as a prefix-free code) can beat the entropy on average. Therefore, $\overline{K}(\mathcal{S}_n) \geq H(\mathcal{S}_n)$.

For the upper bound, we can construct a program for all strings generated by \mathcal{S}_n as follows: The program simply takes n as its first input, enumerates all strings of length n , and computes the probabilities of all these strings (recall that we required the probability distribution to be recursive, so this is possible). From these probabilities, the program then constructs a Huffman code that has average code length within 1 bit of entropy [7]. Note that this program does not depend on the actual value of n , so the program has constant size for all sources \mathcal{S}_n . Let c' denote the size of this program. Following this program, we encode its input data: a prefix-free encoding of the length n , followed by the Huffman code for the particular string of length n we are encoding. Let $\alpha(x_{i,n})$ be the Huffman code assigned to string $x_{i,n}$ in this manner, so the total size of the program and its input data is $c' + K(n) + |\alpha(x_{i,n})|$. This specific program/input coding strategy upper-bounds the minimum-length program, so

$$\overline{K}(\mathcal{S}_n) \leq \sum_{i=1}^{m_n} p_{i,n}(c' + K(n) + |\alpha(x_{i,n})|) = c' + K(n) + \sum_{i=1}^{m_n} p_{i,n}|\alpha(x_{i,n})| \leq c' + K(n) + H(\mathcal{S}_n) + 1,$$

where the last bound follows from the use of Huffman codes. With $c = c' + 1$, we get exactly the upper bound claimed in the theorem. ■

The upper bound in the preceding theorem is $K(n) + c$, but notice that there is a simple prefix-free encoding of integers such that $K(n) = O(\log n)$. Since \mathcal{S} is a stationary ergodic process, and any such source has a finite entropy rate (see Theorem 4.2.1 in [5]), we know that $H(\mathcal{S}_n)$ grows linearly with n . In particular, if h is the entropy rate of the source \mathcal{S} , then $H(\mathcal{S}_n) \geq (h - \epsilon) \cdot n$ for any $\epsilon > 0$ and sufficiently large n , and so using the upper bound from the theorem above we see that

$$\lim_{n \rightarrow \infty} \frac{K(n) + c}{H(\mathcal{S}_n)} = 0.$$

The obvious consequence is that, in the limit, the two notions of information (Shannon entropy and expected Kolmogorov complexity) agree! We make this explicit in the following corollary.

Corollary 2.5.1. *Given the above definitions,*

$$\lim_{n \rightarrow \infty} \frac{\overline{K}(\mathcal{S}_n)}{H(\mathcal{S}_n)} = 1.$$

2.5.2 Approach 2: Conditional Complexities

Recall the problem we are trying to solve in comparing Kolmogorov complexity to Shannon entropy: If we allow for a constant difference that depends on the probability distribution, how can we be certain that we are not hiding something arbitrarily worse than the entropy itself? In our second approach we solve this by essentially giving the probability distribution “for free,” ensuring that we are examining the information content of the strings generated by the source and not the source itself.

Definition 2.5.1. *The conditional (prefix-free) Kolmogorov complexity of a string x , conditioned on another string y , is denoted $K(x|y)$ and is the minimum-length program that produces x when string y is supplied as an auxiliary input for free.*

Being a little more informal, we can condition on information not expressed as a string, as long as it is clear that this information *can* be encoded in such a fashion. In other words, in the case of interest in this section, we can talk about $K(x|\mathcal{S})$, or the algorithmic prefix complexity of

string x when some description of the source \mathcal{S} (complete with all probabilities) is given for free. To be formal, \mathcal{S} would have to be encoded as a specific string, but we leave such technicalities out of our light coverage in this chapter. We can also define the expected conditional Kolmogorov complexity as

$$\overline{K}(\mathcal{S}|y) = \sum_{i=1}^m p_i K(x_i|y),$$

with $\overline{K}(\mathcal{S}|\mathcal{S})$ following from this definition.

Theorem 2.5.2. *Given the above definitions,*

$$0 \leq \overline{K}(\mathcal{S}|\mathcal{S}) - H(\mathcal{S}) \leq c,$$

where c is a constant that depends only on the descriptive language used for the complexity measure (and in particular, c is independent of the probability distribution of \mathcal{S}).

Proof. The proof of this theorem is in fact very similar to the proof of Theorem 2.5.1, and so we just roughly outline it here. First note that even though we have supplied additional information (on the source) to the calculation of $\overline{K}(\mathcal{S}|\mathcal{S})$, this is still a prefix code and so Shannon's noiseless source-coding theorem gives the lower bound on $\overline{K}(\mathcal{S}|\mathcal{S}) - H(\mathcal{S})$.

For the upper bound, notice that if the “for free” information is a description of the source as an encoding of all (x_i, p_i) pairs, then a program can use this information to create a Huffman code and can decode these codes into the x_i strings. The size of this program is just a constant amount for the program code, plus the space required for the Huffman code for the desired string. Taking the expected value, and the bound for the efficiency of Huffman codes, we get the upper bound stated in the theorem. ■

2.5.3 Discussion

The results given in this section, showing the relationship between Kolmogorov complexity and Shannon Information Theory, are roughly what one would hope for from two measures of the same quantity, in this case information. The difference between the two measures, when examined in two different ways, is at most an additive constant, which is the best that can be hoped for. However, since this relationship is desirable it is tempting to not delve deeper and to miss the strength of these results.

In fact, what these results say is almost miraculous. Consider that we have two parts to our source, the strings (the x_i 's) and the probabilities (the p_i 's), and there is no dependence at all between these two parts. It is even possible to take a source and rearrange the strings so that the strings correspond to entirely different probabilities. Furthermore, our two information measures deal primarily with these two different parts: Kolmogorov complexity is determined primarily from the actual x_i strings, and Shannon's entropy does not even consider the strings! Despite the fact that these two measures look at two different and independent values, they still manage to agree in a very strong way. This goes far beyond “coincidence” and provides additional evidence that we are indeed defining information in a proper and consistent manner.

The existence of a universal information measure, for the most part independent of the probability distribution, is somewhat related to a line of reasoning by Ray Solomonoff, the first inventor of Kolmogorov complexity. In particular, in the area of inductive inference, applying Bayes' Rule one must in some manner decide upon a prior distribution, and Solomonoff demonstrated that, using the ideas of Kolmogorov complexity (although it was not known by this name at the time), there exists a *universal* prior distribution that works as well as the true prior as long as the true

prior is a recursive distribution. This idea of a universal probability distribution is an extremely powerful result of this theory, but is beyond the scope of this chapter—for more information, see the references given in the following section.

2.6 HISTORICAL NOTES

The history of Kolmogorov complexity, or Algorithmic Information Theory, is quite interesting. The same core principles were discovered independently by at least three different people within the span of a few years. The convergence of results and ideas, in areas ranging from computability to probability theory, made the emergence of Kolmogorov complexity inevitable, but the three inventors deserve a great deal of credit for putting the appropriate pieces together.

Historically, the first person to put forward ideas along these lines was Ray Solomonoff in the early 1960s [13]. Solomonoff was studying inductive inference and problems with Bayesian priors in particular. Using the still-new ideas in computability, he developed a theory of “algorithmic probability” to use in his theory of inductive reasoning. While the ideas Solomonoff proposed are essentially the same as what we see in Kolmogorov complexity today, the setting was a bit different (and more specialized) and the results were not widely known at the time.

In the mid-1960s, Kolmogorov complexity as we know it was discovered independently and almost simultaneously by Andrei Kolmogorov and Gregory Chaitin. The setting for Kolmogorov and Chaitin was more similar to the context in which we study Kolmogorov complexity today, with a focus on information and the fundamental question of “what is random data?” Kolmogorov was a very-well-known Russian mathematician with a long and distinguished career, which explains to some extent why his name is associated with the theory. His basic paper [8], while brief, outlines the basic ideas and proves the Invariance Theorem.

Chaitin was the last in the time sequence of inventors, but his story is nonetheless fascinating. Chief among the interesting facts is that Chaitin was only 18 years old at the time of his discoveries, which he submitted as a pair of papers to the prestigious *Journal of the ACM* [1, 2]. Chaitin has continued his career with, among other significant work, a series of book publications exploring the nature of information (and in particular algorithmic information theory), randomness, and incompleteness.

2.7 FURTHER READING

There is a great deal published on Kolmogorov complexity, but the best place to start for more information would be either Li and Vitányi’s book [10] or, for a different take on the issues (albeit with a less broad-ranging coverage), any of Chaitin’s books in this area, such as [3, 4]. The application of these ideas goes far beyond data compression issues, as the notion of “information” is fundamental to many different areas from traditional mathematics, to computer science, to physics and other areas.

2.8 REFERENCES

1. Chaitin, G. J., 1966. On the length of programs for computing finite binary sequences. *Journal of the ACM*, Vol. 13, No. 4, pp. 547–569.
2. Chaitin, G. J., 1969. On the length of programs for computing finite binary sequences: Statistical considerations. *Journal of the ACM*, Vol. 16, No. 1, pp. 145–159.

3. Chaitin, G. J., 1988. *Algorithmic Information Theory*. Cambridge Univ. Press, Cambridge, UK.
4. Chaitin, G. J., 1992. *Information-Theoretic Incompleteness*. World Scientific, Singapore.
5. Cover, T. M., and J. A. Thomas, 1991. *Elements of Information Theory*. Wiley, New York.
6. Hopcroft, J. E., and J. D. Ullman, 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
7. Huffman, D. A., 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Electrical and Radio Engineers*, Vol. 40, No. 9, pp. 1098–1101.
8. Kolmogorov, A. N., 1965. Three approaches to the quantitative definition of information. *Problems of Information Transmission [Translated from Problemy Peredachi Informatsii (Russian)]* Vol. 1, pp. 1–7.
9. Li, M., and P. Vitányi, 1990. Kolmogorov complexity and its applications. In *Handbook of Theoretical Computer Science* (J. V. Leeuwen, Ed.), pp. 187–254, Elsevier Science, Amsterdam.
10. Li, M., and P. Vitányi, 1997. *An Introduction to Kolmogorov Complexity and Its Applications*, 2nd ed. Springer-Verlag, New York.
11. Schneier, B., 1996. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. Wiley, New York.
12. Shannon, C., 1948. The mathematical theory of communication. *Bell System Technical Journal*, Vol. 27, pp. 379–423 and 623–565.
13. Solomonoff, R., 1964. A formal theory of inductive inference (I and II). *Information and Control*, Vol. 7, pp. 1–22 and 224–254.
14. Solovay, R. M., 1975. Lecture Notes on Algorithmic Complexity. Unpublished, University of California at Los Angeles.
15. Zvonkin, A., and L. Levin, 1970. The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms. *Russian Mathematical Surveys*, Vol. 25, pp. 83–124.

PART II

Compression Techniques

This Page Intentionally Left Blank

Universal Codes

PETER FENWICK

3.1 COMPACT INTEGER REPRESENTATIONS

Many data compressors produce as intermediate output one or more sets of integers, with smaller values being most probable and larger values increasingly less probable. The final, coding stage must convert these integers into a bit stream, such that each value is self-delimiting and the total number of bits in the whole stream is minimized.

When only a finite alphabet of numbers must be handled and where the number probabilities are known *a priori* it is appropriate to use the well-known Huffman or Shannon–Fano codes, possibly in an adaptive version which adjusts the code as the probabilities are discovered.

This chapter describes ways of encoding an arbitrary integer, often unbounded in size, in as few bits as possible. A simple algorithm should be able to encode an integer into the output bit stream or recover an integer from an input bit stream, even if that particular value has never been seen before. These codes, self-delimiting and variable in length for different values, are usually known as *Universal Codes* or *Variable-Length Codes*.

Many of these codes are difficult to find in the literature, often being secondary aspects or side issues in papers on other topics. The book by Bell *et al.* [1] has a (very) few pages on the subject. Salomon [2] discusses several codes in passing, but refers only obliquely to the Fibonacci codes.

3.2 CHARACTERISTICS OF UNIVERSAL CODES

Universal codes may have to satisfy several requirements, whose relevance depends on the context. For example, robustness may be quite unimportant in many computer applications with error-free transmission, but may be crucial in noisy radio communications. Transparency is unimportant in most computer work, except that it may facilitate conversion to and from the internal representation of the computer.

For all of these codes, an integer N occurring with probability P_N is encoded into or represented by a *codeword* of L binary digits. A fundamental result of information theory is that for optimal coding all codewords have a length $L = \log_2(1/P_N)$ bits, leading to the important general rule that frequent values should have short codewords and less frequent values should have longer codewords.

Efficiency: An *efficient code*, or *compact code*, is assumed to be one which, for each value, approximates the minimum codeword length in comparison with other codes. In many codes the codeword length is nearly a constant multiple of the binary representation. For these codes the expansion relative to binary is a useful indication of efficiency.

Length indication: Every universal representation must somehow indicate its length or signal the end of the bit stream for the codeword. Some representations are terminated by a special bit pattern or *comma*. Others have an explicit length as part of the representation, the length itself often using one of the terminated representations.

Transparency: It is pleasant for human understanding, but certainly not essential, if the value is easily extracted once a codeword is recognized. Some codes, for example, have a portion of the codeword as the natural binary representation of the value.

Robustness: The increasingly subtle coding techniques used to increase code efficiency often reduce the redundancy of the code. But the sensitivity to transmission errors usually increases with the decreasing redundancy. In extreme cases a single bit error can lead to catastrophic error propagation and loss of codeword demarcation. Codes with a terminating pattern or comma tend to be more robust and those with an encoded length less robust.

Instantaneousness: This describes a code that is self-contained, in that a codeword can be decoded without reference to adjoining codewords. A non-instantaneous code is fine in a long sequence of codewords from the same code, but may be inconvenient if the compact codes are interspersed with other, instantaneous, codes.

As a final introductory note, within this chapter the term *bit* will always mean *binary digit* rather than an information measure. A codeword (the encoded collection of bits to represent the value) is usually longer than the binary representation of the value (also in bits).

3.3 POLYNOMIAL REPRESENTATIONS

Many representations for an integer N combine a visible *digit vector* \mathbf{d} with an implicit *weight vector* \mathbf{w} , such that $N = \mathbf{d} \cdot \mathbf{w}$, the scalar product of the two vectors. A polynomial in some base b results if successive terms of the weight vector are given by $w_{i+1} \approx bw_i$. While an obvious example of a polynomial representation is the familiar decimal representation, it is more instructive to consider some other forms.

Binary: ($b = 2$) The weight vector is powers of 2, least to the right, or $\mathbf{w} = \{\dots, 128, 64, 32, 16, 8, 4, 2, 1\}$ and the visible digits $d_i \in \{0, 1\}$. The (decimal) value 23 is represented by 10111

$$23 = 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1.$$

Ternary: ($b = 3$) The weight vector is now $\mathbf{w} = \{\dots, 729, 243, 81, 27, 9, 3, 1\}$ and $d_i \in \{0, 1, 2\}$. 209_{10} is represented by 21202

$$209 = 2 \times 81 + 1 \times 27 + 2 \times 9 + 0 \times 32 + 2 \times 1.$$

Table 3.1 Effective Bases of Some Codes

Code	Base (b)	Expansion (x)
Unary (α)	1	—
Binary (β)	2	1
Elias γ	$\sqrt{2}$	2
Fibonacci-2	$\frac{\sqrt{5}+1}{2}$	1.440
Ternary	$\sqrt{3}$	1.262
Fibonacci-3	See text	1.137

Fibonacci: This is a less obvious example with the weight vector the Fibonacci numbers $w = \{\dots, 34, 21, 13, 8, 5, 3, 2, 1\}$ and the digit vector d again binary. The (decimal) value 20 is represented by 101010

$$20 = 1 \times 13 + 0 \times 8 + 1 \times 5 + 0 \times 3 + 1 \times 2 + 0 \times 1.$$

As successive Fibonacci numbers tend to the ratio $(\sqrt{5} + 1)/2 \approx 1.618$, this may be regarded as another polynomial representation.

Many universal codes (including the Fibonacci example above) are approximated by polynomial codes with a non-integral b , $1 < b < 2$. The base b representation of N has $\log_b N$ digits d_i (that are necessarily binary). As the binary representation has $\log_2 N$ digits and $b < 2$, the base b representation expands relative to binary. The base b and expansion x are both descriptive of the representation and are related by

$$x = \log 2 / \log b, \quad \text{or} \quad b = 2^{1/x}$$

To anticipate later developments, the bases and expansions of several codes are given in Table 3.1. The simplest of these is the Elias γ code, which, generating two bits per binary digit, is equivalent to a number with a base of $\sqrt{2} = 1.414$. At the other extreme, the Fibonacci-3 (or Tribonacci) numbers have a ratio between successive digit weights of

$$\frac{1 + \sqrt[3]{19 - 3\sqrt{33}} + \sqrt[3]{19 + 3\sqrt{33}}}{3} \approx 1.83928675521,$$

giving an effective number base of about 1.84 and an expansion relative to binary of $\log 2 / \log 1.839 = 1.137$.

Some codes add pairs of bits as the value grows; on average these have a small additive constant when calculating the length. Although their growth may be slower than that of codes that grow 1 bit at a time, the additive term may make them less attractive for small values. Similarly, codes with smaller expansions often need a more complex length indication, which may cancel any benefit from the smaller expansion.

3.4 UNARY CODES

The unary code of an integer N is simply a sequence of N 1's followed by a 0 (sometimes N 0's with a terminating 1). A variant code for N may have $(N - 1)$ leading digits but cannot encode a 0. Despite their apparent triviality, unary codes are important and often components of more complex codes. (The unary code 11...10 is a polynomial code, with base $b = 1$ and terminated by the first 0.)

Unary codes are optimal for highly skewed symbol distributions, where the symbol probabilities are related as $P_{k+1} \leq P_k/2$.

3.5 LEVENSTEIN AND ELIAS GAMMA CODES

These codes were first described by Levenstein [3], but the later description by Elias [4] is generally used in the English language literature. Elias describes a whole series of codes:

alpha code: The $\alpha(N)$ code is one of the unary representations above, with $(N - 1)$ 0's followed by a 1. ($\alpha(N)$ has a length of N digits.) Thus $\alpha(7)$ is 0000001. The α code is an example of a *comma* code, where the comma is the terminating 1.

beta code: The $\beta(N)$ code is the natural binary representation of N , starting from the most significant 1. $\beta(6)$ is 110, and $\beta(19)$ is 10011. Sometimes the β code may be modified by omitting the leading 1 bit. The β code is of little use by itself because it has no length indication.

gamma code: The γ code is an intermingling of the bits of the β code and an α code describing its length, omitting the first bit of the β code (which is always 1). Each numeric bit (from the β code) is preceded by a 0 *flag* bit (from the α code), with the whole terminating in the final 1 from the α code. These codes are shown in the first part of Table 3.2, with the flag bits marked by an overline.

gamma' code: The γ' code is a permutation of the γ code, with the flag bits (now an α code) preceding the data bits (a β code) and the terminating 1 of the α prefix acting also as the leading 1 of the β suffix.

For most of this chapter the term Elias γ code will be used interchangeably for the two variants, often meaning the γ' code. Examples of the four codes (α , β , γ , and γ') are shown in Table 3.3.

Ignoring the terminating condition, the Levenstein and Elias γ codes represent each binary digit by 2 bits; the expansion is $x = 2$ and the effective base is $b = \sqrt{2} \approx 1.414$, as shown in Table 3.1.

The γ code can be extended to higher number bases where such granularity is appropriate. For example, numbers can be held in byte units, with each 8-bit byte containing 1 flag bit (last-byte/more-to-come) and 7 data bits, to give a base-128 code.

Some variants of the γ codes are much older than the systematic descriptions by Levenstein and Elias (1968 and 1975, respectively). For example, the IBM 1620 computer (ca. 1960) used

Table 3.2 Elias' Gamma and Gamma' Codes

γ -Code	γ' Code
$\gamma(1) = \bar{1}$	$\gamma'(1) = \bar{1}$
$\gamma(2) = \bar{0}\bar{1}$	$\gamma'(2) = \bar{0}10$
$\gamma(3) = \bar{0}\bar{1}\bar{1}$	$\gamma'(3) = \bar{0}11$
$\gamma(4) = \bar{0}\bar{0}\bar{0}\bar{1}$	$\gamma'(4) = \bar{0}\bar{0}100$
$\gamma(5) = \bar{0}\bar{1}\bar{0}\bar{0}\bar{1}$	$\gamma'(5) = \bar{0}\bar{0}101$
$\gamma(6) = \bar{0}\bar{0}\bar{0}\bar{1}\bar{1}$	$\gamma'(6) = \bar{0}\bar{0}110$
$\gamma(13) = \bar{0}\bar{1}\bar{0}\bar{0}\bar{1}\bar{1}$	$\gamma'(13) = \bar{0}\bar{0}\bar{0}1101$
$\gamma(23) = \bar{0}\bar{1}\bar{0}\bar{1}\bar{0}\bar{1}\bar{0}\bar{0}\bar{1}$	$\gamma'(23) = \bar{0}\bar{0}\bar{0}\bar{0}10111$
$\gamma(44) = \bar{0}\bar{0}\bar{0}\bar{0}\bar{0}\bar{1}\bar{0}\bar{1}\bar{0}\bar{0}\bar{1}$	$\gamma'(44) = \bar{0}\bar{0}\bar{0}\bar{0}\bar{0}101100$

Table 3.3 Examples of Elias α , β , and γ Codes

N	$\alpha(N)$	$\beta(N)$	$\gamma(N)$	$\gamma'(N)$
1	1	1	1	1
2	01	10	001	010
3	001	11	011	011
4	0001	100	00001	00100
5	00001	101	01001	00101
6	000001	110	00011	00110
7	0000001	111	01011	00111
8	00000001	1000	0000001	0001000
9	000000001	1001	0100001	0001001
10	0000000001	1010	0001001	0001010
50	...	110010	00010000011	00000110010

BCD coding with a flag bit on each decimal digit (memory addressed by individual decimal digit). Numbers were addressed at the least-significant digit and proceeded to lower addresses until terminated by a flagged digit—a precise implementation (or anticipation) of a BCD variant of the γ code. (A flag on the addressed digit denoted a negative number.)

As every *numeric* bit requires two *codeword* bits the γ codes require 2 bits to double the range of values and are equivalent to a polynomial representation with base $b = \sqrt{2}$. The γ codes are very efficient for very small values (they are one of the few codes that represent the smallest value with a single bit). For larger values the large number of flag bits becomes an ever-increasing overhead, as shown by their relatively large expansion.

3.6 ELIAS OMEGA AND EVEN–RODEH CODES

With the γ code length given as a unary code, a natural progression is to encode the length itself as γ code. Elias does this with his δ code, using a γ code for the length, but quickly proceeds to ω codes. Some very similar codes were described by Even and Rodeh [5] and it is convenient to treat the two together. Each of the codes has the value (as a β code) preceded by its length in binary. If the length is not representable in 2 bits (ω code; 3 bits for Even–Rodeh), encode a prefix giving the “length of the length,” recursing until a 2-bit value is obtained.

Elias ω code: Some Elias ω codes are shown in Table 3.4, with groups of bits separated by blanks. Each length is followed by the most-significant 1 of the next length or value; the final value is followed by a 0. The codes are most easily described by giving the decoding process, unwinding the recursive generation of the code value.

The first group of bits is always either 10 or 11, specifying a value of 2 or 3, or a following group of 2 or 3 bits. If a group in an ω code is followed by a 0, its value is the value to be delivered. If a group is followed by a 1, its value is the number of bits to be read and placed after that following 1 to give the value of the next group. Thus 15 is read as the sequence 3, 15 and 16 is read as 2, 4, 16. (Observe that the initial bits of each group form an α code controlling the codeword length.) As all other codewords start with a 1, a single 0 bit is used to denote a value of 1.

Even–Rodeh code: The Even–Rodeh code is similar to the Elias ω code, but each group now gives the total number of bits in the following group, *including* the most significant 1. A different starting procedure is used, with values of 0–3 written as 3-bit integers and the

Table 3.4 Examples of Elias' ω and Even–Rodeh Codes

Value	Elias ω code	Even–Rodeh Code
0	—	000
1	0	001
2	10 0	010
3	11 0	011
4	10 100 0	100 0
7	10 111 0	111 0
8	11 1000 0	100 1000 0
15	11 1111 0	100 1111 0
16	10 100 10000 0	101 10000 0
32	10 101 100000 0	110 100000 0
100	10 110 1100100 0	111 1100100 0
1000	11 1010 1111101000 0	1010 1111101000 0

Table 3.5 Lengths of Elias' ω and Even–Rodeh Codes

Values	Elias	Even–Rodeh
1	1	3
2–3	3	3
4–7	6	4
8–15	7	8
16–31	11	9
32–63	12	10
64–127	13	11
128–255	14	17
256–512	21	18

initial group now 3 bits. Values of 4–7 are in natural binary, with a following 0 as terminator. The Even–Rodeh code *does* represent a 0 value, as the bits 000; the smallest value for the Elias ω code is 1.

Both codes are especially efficient just before a new length element is phased in and inefficient just after it is introduced, as for 15 and 16 in the Elias ω code. The codes alternate in relative efficiency as their extra length components phase in at different values, as shown in Table 3.5.

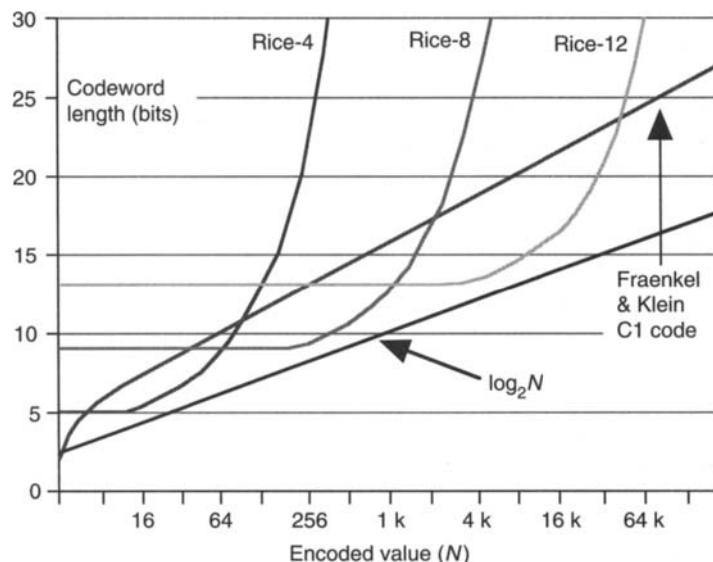
Bentley and Yao [6] develop a very similar code as a consequence of an optimal strategy for an unbounded search, recognizing a correspondence between the tests of the search and the coding of index of the search target, but do not develop the code to the detail of either Elias or Even and Rodeh.

3.7 RICE CODES

Rice codes [7] have an integer parameter k . The representation is the concatenation of $(1 + N/2^k)$ as a unary code and $(N \bmod 2^k)$ in binary. (Using $1 + N/2^k$ allows for $N < 2^k$.) An integer N is represented by about $N/2^k + k + 1$ bits. Representative Rice codes are shown in Table 3.6.

Table 3.6 Rice Codes for the First Few Integers and Parameter k

k	0	2	3	4
0	0	000	0000	00000
1	10	001	0001	00001
2	110	010	0010	00010
3	1110	011	0011	00011
4	11110	1000	0100	00100
5	111110	1001	0101	00101
6	1111110	1010	0110	00110
7	11111110	1011	0111	00111
8	111111110	11000	10000	01000
9	1111111110	11001	10001	01001
10	11111111110	11010	10010	01010
11	111111111110	11011	10011	01011
12	1111111111110	111000	10100	01100
13	11111111111110	111001	10101	01101
14	111111111111110	111010	10110	01110
15	1111111111111110	111011	10111	01111

**FIGURE 3.1**
Lengths of Rice codes (smoothed).

The Rice codes are extremely efficient for values of N near 2^k as shown in Fig. 3.1, comparing Rice codes for $k = 4, 8$, and 12 with the Fraenkel and Klein $C1$ code. The $C1$ code is generally one of the shorter codes; within the appropriate ranges the Rice codes may be several bits shorter. (The codeword lengths should show discontinuities as each bit is introduced but these steps have been smoothed here to make the graphs easier to understand.) The graph also shows how closely the Rice codes approach $\log_2 N$, the binary code length or least possible length.

As smaller values are always represented by $k + 1$ bits, the code becomes progressively less efficient for small values, with many leading 0's. Larger values quickly become less efficient because the α code prefix grows linearly with N , appearing as an exponential increase with the logarithmic axis of Fig. 3.1.

The Rice codes were first developed for deep-space telemetry and especially video transmission. Much of this video data is characterized by a relatively small range of numerical values and Rice shows that by appropriate selection of the parameter k it is possible to code with very small redundancy. The use of the Rice codes in the lossless data compression standard for space applications is described in Chapter 16.

3.8 GOLOMB CODES

The Golomb codes [8] are designed to encode a sequence of asymmetric binary events, where a probable event with probability p is interspersed with unlikely events of probability q ($q = 1 - p$ and $p \gg q$).¹

The sequence is represented by the lengths of successive runs of the probable event between occurrences of the improbable event. The Golomb codes have a parameter m , related to the probability p by $p^m = 0.5$, or $p = \sqrt[m]{0.5}$. A run of length $N + m$ is half as likely as a run of length N , indicating that the codeword for a run of length $N + m$ should be 1 bit longer than that for a run of length N .

Golomb codes may be regarded as a generalization of the Rice codes, with a Rice(k) code identical to the Golomb (2^k) code. They are the most complex of the codes of this chapter, with quite different rules for encoding and decoding.

The dictionary for a Golomb (m) code consists of a sequence of codeword groups, numbered from 0. All groups except the first have m codewords all of the same length, starting with an α prefix. Each codeword within a group is obtained by adding 1 to its predecessor, but this addition may overflow into the α prefix and modify it for later members of the group. The first member of a group has the last value from the previous group incremented by 1 and extended by a single trailing 0. The dictionary starts with a small group of shorter words.

This example assumes that all arithmetic is in binary or, rather, that the final codeword is represented in binary. To encode N with the Golomb (m) code, first find the least k , such that $2^k \geq 2m$:

$$k = \lceil \log_2 m \rceil + 1. \quad (3.1)$$

The dictionary starts with a small group of j codewords of length $(k - 1)$ bits

$$j = 2^{k-1} - m. \quad (3.2)$$

Calculate g , the number of the group which encodes N :

$$g = (N - j)/m + 1. \quad (3.3)$$

Now generate a raw codeword p with $(g - 1)$ leading 1's and k following 0's:

$$p = (2^{g-1} - 1)2^k. \quad (3.4)$$

The first or *base* codeword b of group g is

$$b = p + 2j. \quad (3.5)$$

¹ Golomb's original paper should be consulted for a full description of how the code developed from activities of Secret Agent 0 0 111 at the casino!

Find the first value s which will encode into the group

$$s = m(g - 1) + j \quad (3.6)$$

and add the difference $(N - s)$ into the base codeword, obtaining the final codeword w

$$w = b + (N - s). \quad (3.7)$$

Ignoring the special case of the first group, the first codeword of a group represents the value $(m(g - 1) + j)$ and starts with an α code prefix $\alpha(g - 1)$ followed by the value $2j$ to $(k - 1)$ bits. Later values within the group of m codewords are obtained by successively incrementing the predecessor code. Because the counting eventually overflows into the end of the α code prefix, the group always ends with several words that *seem* to belong to the next group. These anomalous codewords complicate decoding.

The $m = 6$ code is shown in Table 3.7, with a colon separating the prefix and suffix of each codeword. Groups are separated by a horizontal line; there is a group boundary between 7 and 8, but not at other column boundaries. The overflow into the α prefix is seen in the codewords for 6, 12, 18, 24, and 30.

The Golomb coder produces groups of codewords with m words of each length. Most codewords also occur in groups with 1, 2, 3, ... leading 1's, with m words to a group. Unfortunately the encoding and decoding groups do not coincide unless $m = 2^k$.

If m is a power of 2 (corresponding to a Rice code), the codeword for N is a simple concatenation of $\alpha(1 + N/m)$ as a prefix, followed by the binary representation of $N \bmod m$ to $\log m$ bits, i.e.,

$$\alpha(1 + N/m) : \beta(N \bmod m).$$

Table 3.8 shows a few Golomb codes for this case, with the α and β components separated by a colon for the special case where $2^k = 2m$ and $j = 2^{k-1} - m = 0$. Some general examples of Golomb codes are shown in Table 3.9.

Table 3.7 Structure of the Golomb(6) Code

0 :000	8 10:100	16 110:110	24 1111:000
1 :001	9 10:101	17 110:111	25 1111:001
2 0:100	10 10:110	18 111:000	26 11110:100
3 0:101	11 10:111	19 111:001	27 11110:101
4 0:110	12 11:000	20 1110:100	28 11110:110
5 0:111	13 11:001	21 1110:101	29 11110:111
6 1:000	14 110:100	22 1110:110	30 11111:000
7 1:001	15 110:101	23 1110:111	31 11111:001

Table 3.8 Some Golomb Codes for $m = 2^i$

$m \rightarrow$ $\downarrow N$	2	4	8
3	10:1	0:11	0:011
9	11110:1	110:01	10:001
14	11111110:0	1110:10	10:110

Table 3.9 Golomb Codes for Small Integers, with Parameter m

$m \rightarrow$ $\downarrow N$	1	2	3	4	5
0	0	00	00	000	000
1	10	01	010	001	001
2	110	100	011	010	010
3	1110	101	100	011	0110
4	11110	1100	1010	1000	0111
5	111110	1101	1011	1001	1000
6	1111110	11100	1100	1010	1001
7	11111110	11101	11010	1011	1010
8	111111110	111100	11011	11000	10110
9	1111111110	111101	11100	11001	10111
10	11111111110	1111100	111010	11010	11000
11	111111111110	1111101	111011	11011	11001
12	1111111111110	11111100	111100	111000	11010

To decode the Golomb (m) code, the parameter m immediately gives the values of k and j from Eqs. (1) and (2) above. If the first digit is 0, then either $g = 0$ (digits 00...) or $g = 1$ (digits 01...). After the a leading 1's, read the next $(k - 1)$ bits (starting with the first 0) as a value x . If $x \geq j$, then read one more input digit to the low-order end of x . If $x < j$, reduce the value a by 1 because this codeword really belongs to the preceding coding group.

Using the symbols from the code generation, reverse the encoding process, to give

$$N = [m(a - 1) + j] + [x - 2j].$$

The comments on the Rice code efficiency apply mostly to Golomb codes, remembering the equivalence of a Golomb (2^k) and a Rice (k) code. A Golomb code shows the same run-away codeword length for large values, and high efficiency for, say, $m/2 < N < 2m$. However, if $m \neq 2^k$, a Golomb code is shorter than a Rice code for very small values, giving an advantage over the Rice codes.

3.9 START-STEP-STOP CODES

These codes [9] are defined by three parameters, i , j , and k . The representation may be less clearly related to the value than for Elias γ and Rice codes. If the last parameter k is finite, these codes handle only a finite alphabet; an infinite alphabet requires that $k = \infty$.

The codewords contain both a prefix and a suffix. The code defines a series of blocks of codewords of increasing length, the first block with a suffix of i bits (β code), the second with $i + j$ bits, then $i + 2j$ bits, and so on, up to a final suffix length of k bits. A unary prefix gives the number of the suffix group. Thus a 3, 2, 9 code has codewords with suffixes of 3, 5, 7, and 9 bits and prefixes of 0, 10, 110, and 111 (omitting the final 0 from the last prefix) as shown in Table 3.10.

The start-step-codes can generate many of the other codes, or codes equivalent to them, as shown in Table 3.11.

Table 3.10 Code Values for a 3, 2, 9 Start–Step–Stop Code

	Codeword	Range
	0xxx	0–7
	10xxxxxx	8–39
	110xxxxxxxx	40–167
	111xxxxxxxxx	168–679

Table 3.11 Special Cases of Start–Step–Stop Codes

Parameters			Generated Code
i	j	k	
N	1	N	A simple binary coding of the integers to $2^N - 1$
0	1	∞	The Elias γ' code
N	N	∞	The base 2^N Elias γ' code
N	0	∞	A code equivalent to the Rice(N) code

Table 3.12 The First Few Fibonacci Numbers

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}
1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1,597	2,584

3.10 FIBONACCI CODES

Universal codes based on Fibonacci numbers were first described by Apostolico and Fraenkel [11] and later by Fraenkel and Klein [12]. Both emphasize the robustness of the codes or their ability to recover from bit errors. Here the emphasis is on their use as universal or variable-length codes, assuming error-free transmission. The presentation here follows the development of the second (Fraenkel and Klein) paper as being rather easier to understand. The earlier paper by Apostolico and Fraenkel treats more general cases and misses some simpler aspects; its results will be given later.

The Fibonacci numbers are a famous integer sequence where each number is the sum of its two immediate predecessors. The first few values are shown in Table 3.12.

$$F_i = F_{i-2} + F_{i-1}, \quad \text{where } F_1 = F_2 = 1. \quad (3.8)$$

The ratio of successive Fibonacci numbers tends to the *golden ratio* φ .

$$\frac{F_{k+1}}{F_k} = \varphi = \frac{1 + \sqrt{5}}{2} \approx 1.618034. \quad (3.9)$$

3.10.1 Zeckendorf Representation

A standard result of Fibonacci theory is Zeckendorf's theorem [10, p. 108], which states that any integer can be formed as the sum of Fibonacci numbers. This *Zeckendorf representation* is coded by writing a binary vector with a 1 wherever that Fibonacci number is included, but omitting F_1 as redundant; see Vajda [10] or Knuth [13, p. 85, Ex. 34].

Thus $19 = 13 + 5 + 1$; its Zeckendorf representation is $Z(19) = 101001$ because $19 = 1 \times 13 + 0 \times 8 + 1 \times 5 + 0 \times 3 + 0 \times 2 + 1 \times 1$. The standard Zeckendorf representation assumes that the least-significant bit is on the right, in line with normal numerical representations. When coding it is often better to write the bits in the reverse order, least-significant first, which is represented as $F(N)$. Thus $F(19) = 100101$.

The crucial property of the Zeckendorf representation is that it never has two adjacent 1's. By the definition of the Fibonacci numbers (Eq. (3.8)), such a pair is always equivalent to a single more-significant 1.

As the ratio of two successive Fibonacci numbers tends quickly to $\varphi \approx 1.618034$ (Eq. (3.9)), codes based on the Fibonacci numbers are polynomial representations with a base $b \approx 1.618$ and an expansion $x = 1.440$, as shown earlier in Table 3.1.

3.10.2 Fraenkel and Klein Codes

The Fraenkel and Klein C^1 code (which will be called the FK_1 code) follows immediately from the property that no legitimate Zeckendorf representation has two consecutive 1's. It is simply $F(N)$ immediately followed by another 1 as a terminating comma. Thus, $FK_1(19) = 1001011$.

Fraenkel and Klein develop two other related codes. For uniformity, assume that the smallest value for all codes is 1.

1. The FK_2 code (their C^2 code) includes only $F(N)$ representations with a least-significant bit of 1. This results in codewords of the form $10\dots1$. Alternatively, the codeword for N can be obtained by adding a prefix 10 to the representation of $F(N - 1)$. As a special case, the value 1 is represented by a single 1 bit. In contrast to the FK_1 codes, the FK_2 codes are not instantaneous, as each codeword overlaps its neighbors. However, the FK_2 code does allow a more compact representation for the smallest value.
2. The C^3 code (FK_3) is obtained by taking all values of $F(N)$ of some length, say, r bits, and writing down the block twice, first with a prefix of 10 and then with a prefix of 11. Every codeword of FK_3 has an initial 1 bit, no codeword has more than 3 consecutive 1's (and any consecutive 1's appear only as a prefix), and every codeword except $FK_3(2)$ terminates in 01.

Some examples of these codes are shown in Table 3.13, together with the codeword lengths. There is relatively little difference between them; all are better for some values and worse for others. Measurements by Fraenkel and Klein on recoding simple English text show that FK_1 and FK_3 give very similar performances and are superior to FK_2 . But FK_2 has a shorter representation for 1 and may be better for more highly skewed distributions.

3.10.3 Higher-Order Fibonacci Representations

Traditional Fibonacci numbers involve the sum of *two* predecessors. In an order- m Fibonacci sequence each number is the sum of its m predecessors. (The order-3 numbers are often known as the *Tribonacci* numbers.) The first few Fibonacci numbers of orders 2 and 3 are shown in

Table 3.13 Fraenkel and Klein's Codes

N	Codewords			Lengths			
	FK ₁	FK ₂	FK ₃	FK ₁	FK ₂	FK ₃	
1	11		1	101	2	1	3
2	011		101	111	3	3	3
3	0011		1001	1001	4	4	4
4	1011		10001	1101	4	5	4
5	00011		10101	10001	5	5	5
6	10011		100001	10101	5	6	5
7	01011		101001	11001	5	6	5
8	000011		100101	11101	6	6	5
9	100011		1000001	100001	6	7	6
10	010011		1010001	101001	6	7	6

Table 3.14 Fibonacci Numbers of Orders 2 and 3

$F_1^{(2)}$	$F_2^{(2)}$	$F_3^{(2)}$	$F_4^{(2)}$	$F_5^{(2)}$	$F_6^{(2)}$	$F_7^{(2)}$	$F_8^{(2)}$	$F_1^{(3)}$	$F_2^{(3)}$	$F_3^{(3)}$	$F_4^{(3)}$	$F_5^{(3)}$	$F_6^{(3)}$	$F_7^{(3)}$	$F_8^{(3)}$
1	1	2	3	5	8	13	21	1	1	2	4	7	13	24	44

Table 3.14. The left side is identical to part of Table 3.12, but with the order included in the Fibonacci notation.

These higher-order Fibonacci numbers can be used to generate higher-order analogs of the Zeckendorf representation, with the property that they have no runs of k consecutive 1's if $k \geq m$. Thus an order-3 representation has no runs of 3 or more 1's. Assume throughout this section that the Fibonacci numbers F_k are of order-3, unless otherwise stated.

A simple order- m code for N is simply $Z(N)$ (bits in either order), followed by a 0 and then m 1's. This code is not that efficient, but is useful as an introduction to the better codes of the following sections.

3.10.4 Apostolico and Fraenkel Codes

Apostolico and Fraenkel [11] develop several codes using the higher-order Fibonacci numbers. Their emphasis is not so much on universal codes per se but rather on codes that are robust under occasional data corruption. The discussion here is restricted to order-3 representations, whereas they consider the general case of order- m codes.

They describe two codes, a C_1 code (here AF_1) and then a C_2 code (here AF_2), which is simpler than the AF_1 code. The description here is slightly different from theirs; the original paper should be referred to for much of the underlying theory and justification.

Apostolico and Fraenkel assume that $F_1 = 2$ for all $m \geq 2$. As the *Fibonacci Association* convention is that $F_1 = F_2 = 1$ and $F_3 = 2$ for $m = 2$, the discussion here assumes that all Fibonacci sequences start with $\{F_1 = 1, F_2 = 1, F_3 = 2, \dots\}$, preceded by an appropriate number of 0's for $m > 2$.

Table 3.15 shows examples of the Apostolico and Fraenkel codes (their C_2 and C_1 codes, here AF_2 and AF_1 codes, for order-3) and some new Fibonacci codes from Section 3.10.5.

Table 3.15 Apostolico and Fraenkel's Codes, with New Fibonacci Codes

N	Apostolico and Fraenkel		New Fibonacci Codes	
	AF ₂ Order-3	AF ₁ Order-3	Order-2	NF ₃ Order-3
1	11	111	11	111
2	1011	0111	011	01111
3	10011	00111	0011	11110
4	11011	10111	1011	001110
5	100011	000111	00011	101110
6	101011	010111	10011	01111
7	110011	100111	01011	0001110
8	1000011	110111	000011	1001110
9	1001011	0000111	100011	0101110
10	1010011	0010111	010011	1101110
11	1011011	0100111	001011	001111
12	1100011	0110111	101011	101111
13	1101011	1000111	0000011	00001110
14	10000011	1010111	1000011	10001110
15	10001011	1100111	0100011	01001110
16	10010011	00000111	0010011	11001110

Table 3.16 Development of the Apostolico–Fraenkel AF₁ Codes

k	1	2	3	4	5	6	7	8
F _k	1	1	2	4	7	13	24	44
S = $\sum F_k$	1	2	4	8	15	28	52	96
Range	—	—	3–4	5–8	9–15	16–28	29–52	53–96

3.10.4.1 The Apostolico–Fraenkel AF₂ Codes

These codes represent the value 1 with $(m - 1)$ consecutive 1's. Larger values are encoded as the Zeckendorf representation $Z(N - 1)$, most-significant bit leading, followed by a suffix of 0 and then $(m - 1)$ 1's. (The termination comes from these 1's and the first 1 of the next codeword; the code is not instantaneous.)

3.10.4.2 The Apostolico–Fraenkel AF₁ Codes

These codes involve a transformation or mapping to remove many awkward codewords. To generate the order-3 codes start with the order-3 Fibonacci numbers $F_k^{(3)}$ from Table 3.14 and calculate the cumulative sums $S = \sum F(k)$ of those numbers, as shown in Table 3.16.

To encode a value N :

1. Find k such that $S_{k-1} < N \leq S_k$.
2. Find $Q = N - S_{k-1} - 1$.
3. Encode $F_{k+1} + Q$ in an order-3 Zeckendorf representation, most-significant bit first.
4. Delete the leading 10 from this codeword and attach the suffix 0111 as terminator. (The codeword *always* has a prefix of 10, by virtue of step 2.)

The values 1 and 2 have the special codewords 1 → 111 and 2 → 0111.

Table 3.17 A Range of Order-3 Zeckendorf Representations

N	Digit Weights					N	Digit Weights				
	1	2	4	7	13		1	2	4	7	13
13	0	0	0	0	1	19	0	1	1	0	1
14	1	0	0	0	1	20	0	0	0	1	1
15	0	1	0	0	1	21	1	0	0	1	1
16	1	1	0	0	1	22	0	1	0	1	1
17	0	0	1	0	1	23	1	1	0	1	1
18	1	0	1	0	1						

To encode the value 11 (their example):

1. Find k such that $S_{k-1} < 11 \leq S_k$; $k = 5$, $S_{k-1} = 8$.
2. $Q = 11 - 8 - 1 = 2$.
3. Encode $F_{k+1} + Q = 13 + 2 = 15$ in an order-3 Zeckendorf representation, giving 10 010.
4. Delete the leading 10 and add the suffix 0111 to give 010 0111 as the final codeword.

Again, to encode 40, first find $N = 7$ and $Q = 11$, and then encode $44 + 11 = 55$ to give first 1001100 and thence 01100 0111 as the final codeword.

The second step ($Q = N - S_k - 1$) needs explanation. Consider the order-3 representations of $F_k \leq N < F_{k+1}$ as shown in Table 3.17 for the range $13 \leq N < 24$ (i.e., $F_6 \leq N < F_7$), with the digit weights in the first row and greater weights to the right. By the Fibonacci definitions, there are $(F_{k-1} + F_{k-2})$ values in this range; the F_{k-1} smaller ones end with ...01 and the F_{k-2} larger with ...11. The adjustment $Q = N - S_k - 1$ eliminates all representations with most-significant bits ...11. Thus a received bit sequence ...0111 always corresponds to the numeric bits ...01.

The AF_1 code is then just 2 bits longer than the Zeckendorf representation, whereas the simpler AF_2 code is 3 bits longer than the Zeckendorf representation after absorbing the most-significant 1 bit into its terminator 1110. By discarding some of the possible codewords the AF_1 code is slightly longer for larger values. Against this it is inherently 1 bit shorter than the simpler code; the two effects largely cancel.

From Table 3.1, the Fibonacci-3 (Tribonacci) codes have an effective base $b = 1.839$ and an expansion $x = 1.137$. But they follow the general trend that a more compact code (smaller x) tends to have a more complex length indication that may offset (or even overwhelm) the smaller expansion.

The first order-3 Zeckendorf code above presented the bits in increasing significance, followed by the suffix 0111. In the AF_1 code the bits are presented in *decreasing* significance, again with a suffix 0111. Now consider the AF_1 code with its bits in *increasing* significance, so that the code for 11 is 010 0111 and for 40 is 11010 0111. But these codes are, respectively, 01001 11 and 1101001 11, shifting the break to give a different emphasis to the two components. As both are the representations of $F_{k+1} + Q$ with a suffix of 11, an alternative way of generating a code equivalent to the AF_1 code is to generate $Z(F_{k+1} + Q)$, least-significant bit first, and append a suffix 11.

3.10.5 A New Order-3 Fibonacci Code

In contrast to the rest of the chapter, this section introduces a new, unpublished code as an alternative to existing codes.

Table 3.18 Terminators for Order-5 Fibonacci Code

Numerical Bits	Final Code
...01	...01.1111.0
...011	...011.111.10
...0111	...0111.11.110
...01111	...01111.1.111

As the order-3 Fibonacci codes may end with either one or two consecutive 1's, the terminator must allow the two cases to be distinguished. Apostolico and Fraenkel solve the problem by eliminating those codes whose Zeckendorf representations end in ...11.

For the new code (the NF_3 code), transmit the order-3 Zeckendorf representation least-significant bit first (using $F(N)$) and follow its most significant 1 bit with a suitable comma or terminator as described later. With the order-2 code, the most-significant bit pattern (LSB first) is always ...01 and a single 1 bit acts as an unambiguous terminator. With the order-3 code the most significant bit pattern may be either ...011 or ...01. The terminator must be a run of 111 but it is also necessary to decide how many of those 1's have numeric significance.

The NF_3 code uses the following rules for the terminator:

- If $F(N)$ ends with ...01, add the terminator 110, so that the codeword ends with ...01 110.
- If $F(N)$ ends with ...011, add the terminator 11, so that the codeword ends with ...011 11.

The bit immediately after the terminating sequence 111 indicates how many numeric 1's to retain. But the isolated terminator 111 can be retained as a unique representation for the minimum value. It always follows immediately from another terminator, occurs at the start of the codeword, and can be decoded without ambiguity. While the codeword lengths are similar to those of the Apostolico and Fraenkel C_2 code (a few are 1 bit shorter), the NF_3 code is much simpler to generate. Examples of the NF_3 code were shown earlier in Table 3.15.

The principle can be extended to higher-order Fibonacci codes, but with increasingly expensive terminators. An order- m code must be built out to have m terminating 1's, but then needs a code to say how many of those 1's are numerically significant. The result is that an order-5 code needs on average a 5-bit terminator, as shown in Table 3.18, with dots inserted to separate the components of the terminator. The terminator lengths in Table 3.18 are Huffman coded according to their probabilities. While the order-5 code is inherently quite efficient, its costly terminator means that it is shorter than the order-3 Fibonacci code only for values over about 1 million. Fibonacci codes of order greater than 3 are expected to be useful only in special circumstances.

3.11 TERNARY COMMA CODES

Fenwick [14] shows that bit-pairs can represent the values {0, 1, 2, comma}, leading to a base-3 or ternary code with the digit 3 reserved as a *comma* or terminating code. Table 3.19 shows the ternary comma code representation for the first few integers and some larger ones, with “c” representing the comma. (Although the comma “c” is encoded as the digit 3 or bits 11, it is represented by c to emphasize its non-numerical nature.)

The comma principle can be extended to larger number bases, by sacrificing one of the digits to use as the comma, but becomes increasingly inefficient for small values because the comma consumes a large amount of code space but conveys little information.

Table 3.19 Ternary Codes for the Various Integers

Value	Code	Bits	Value	Code	Bits
1	1c	4	11	102c	8
2	2c	4	12	110c	8
3	10c	6	13	111c	8
4	11c	6	14	112c	8
5	12c	6	15	120c	8
6	20c	6	16	121c	8
7	21c	6	17	122c	8
8	22c	6	18	200c	8
9	100c	8	19	201c	8
10	101c	8	20	202c	8
64	2101c	10	1000	1101001c	16
128	11202c	12	3000	11010010c	18
256	100111c	14	10000	111201101c	20
512	200222c	14	65536	10022220021c	24

Each *ternary* digit of the input value requires 2 bits, giving an effective base $b = \sqrt{3}$ and an expansion relative to natural binary of 1.262, as shown in Table 3.1. To this must be added the constant overhead of 2 bits for the terminating comma.

If the ternary digits are transmitted most-significant first, the first digit must be a 1 or 2 and never 0 or c. Thus a single 0 (bits 00) may be used as a 2-bit representation for a 0 value. An initial c (bits 11) may be used as an escape code or to represent the value 1 (and encoding $N - 1$ for larger values).

3.12 SUMMATION CODES

This class of codes starts with a set of integers and represents values as the sum of a fixed number of members of this set; it is fully described by Fenwick [16].

Just as the binary representation is based on powers of 2 and the Zeckendorf representation on Fibonacci numbers, the summation codes are, initially at least, based on the prime numbers, using the Goldbach conjecture that all even numbers are the sum of two primes.

The Elias α codes have the termination condition *stop at the first 1 bit*. The summation codes may be regarded as a generalization of the α codes and stop at the *second 1 bit*.

The first code is a very simple one, but one that is surprisingly efficient for small values. For later reference it will be known as the G_0 code. As the sum of the primes is always even, encode twice the represented value, for example, 5 as $10 = 3 + 7$. We use the slightly modified table of primes shown in Table 3.20 which forces all sums to be even by omitting the usual value 2. Including a 1 in the set of primes does allow 2 and 3 to be encoded, but adds an extra bit to *all* other codeword lengths and is, overall, not justified.

Then 5 is represented as 101 and 14 ($28 = 11 + 17$) as 000101. The first few codewords are shown in Table 3.21. As very small values are not representable at all, it is usually necessary to encode with an offset.

For small values (essentially those shown), the G_0 code is one of the shortest codes, although the α coding of the first prime index makes it less efficient for values much beyond 15.

Table 3.20 The First Few Prime Numbers, and Indices

Value	3	5	7	11	13	17	19	23
Index	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Value	29	31	37	41	43	47	53	59
Index	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}

Table 3.21 Simple Goldbach Summation (G_0) Codes

N	Sum	Code	N	Sum	Code
1	—	—	9	$7 + 11$	0011
2	—	—	10	$7 + 13$	00101
3	—	—	11	$5 + 17$	010001
4	$3 + 5$	11	12	$11 + 13$	00011
5	$3 + 7$	101	13	$7 + 19$	0010001
6	$5 + 7$	011	14	$11 + 17$	000101
7	$3 + 11$	1001	15	$13 + 17$	000011
8	$5 + 11$	0101	16	$13 + 19$	0000101

3.12.1 Goldbach G_1 Codes

The simple summation code used a direct encoding of the bit positions, but suffered because of the lengths of the 0 runs. Larger values are handled by encoding the lengths of the two 0 runs as γ codes (which are the shortest for small values or short runs). For example, as $20 = 7 + 13$, represent 20 as the two indices 4 and 6, or 4 and $(6 - 4)$. Encoding these two components as a γ' code gives 0010 010 as the representation of 20.

But this is not the only representation, as shown in Table 3.22. For many encoded values it is necessary to choose the combination of primes to give the shortest representation.

This G_1 code is limited to even integers (by the Goldbach conjecture itself), although like the G_0 code it is possible to encode twice the value. Fenwick then describes a modification that encodes any integer. Assuming that p and q are the indices of the primes which add to the integer N , there are several cases to consider. The values p and $q - p$ are both incremented by 1 and then encoded as a γ' code.

Small values: Encode 1 as 110 and 2 as 111.

Even integer: This the sum of the two primes, P_p and P_q . The value is encoded as the concatenation of $p + 1$ and $q - p + 1$.

Prime integer (odd): Here the integer $N = P_i$; emit i (as a γ code) and then a single 1 bit.

Odd integer (non-prime): Emit an initial 1 bit and the i , where $N - 1 = P_i$.

The resultant Goldbach codes are competitive for a range of values, approximately $4 < N < 100$.

3.12.2 Additive Codes

After describing the Goldbach codes, Fenwick develops a family of related codes where all integers within a range can be represented as the sum of two members of a set of carefully selected *basis integers*. The integer is then encoded as the concatenation of the γ' codes of the two components,

Table 3.22 Some Goldbach G_1 Representations, as Sums of Primes

Value	Sums	Indices	Encode As	Coding	Length
18	1 + 17	1,7	1,7	1 00111	6
...	5 + 13	3,6	3,4	011 00100	8
...	7 + 11	4,5	4,2	00100 010	8
20	1 + 19	1,8	1,8	1 0001000	8
...	3 + 17	2,7	2,6	010 00110	8
...	7 + 13	4,6	4,3	00100 011	8
36	5 + 31	3,11	3,9	011 0001001	10
...	7 + 29	4,10	4,7	00100 00111	10
...	13 + 23	6,9	6,4	00110 00100	10
...	17 + 19	7,8	7,2	00111 010	8
42	1 + 41	1,13	1,13	1 0001011	8
...	5 + 37	3,12	3,10	011 0001010	10
...	11 + 31	5,11	5,7	00101 00111	10
...	13 + 29	6,10	6,5	00110 00101	10
...	19 + 23	8,9	8,2	0001000 010	10

Table 3.23 Basis Integers for Additive Code Add₂₅₆

0	1	3	5	7	9	10	21
23	25	27	28	39	41	43	45
47	50	58	76	87	98	102	106
118	120	122	124	135	154	166	183
202	214	231	250				
36 values (54 primes) Seeds = 10, 28, 50							

as for the first Goldbach code. A typical set of basis integers to encode integers up to 256 is shown in Table 3.23.

The Additive code Add_m is designed to encode integers $N \leq m$ (and might not encode $N > m$). The set of basis integers is generated by a sieve, recording the values that can be generated by adding members of the set-so-far. A value which cannot be formed is included as the next member of the basis set. Suitable seeds are introduced to facilitate code generation; seed values are found by a computer search to minimize, for example, the average length of the generated code.

3.13 WHEELER 1/2 CODE AND RUN-LENGTHS

The codes of this section are embedded in a character stream and encode the length of run-length compressed strings.

A simple method, used in some Burrows–Wheeler compressors, triggers run-length encoding by a succession of say four identical characters and then encodes the length in a base-64 γ code. For example, a run of 50 A's would have the hexadecimal coding “41 41 41 41 72” and 1000 X's “58 58 58 58 28 4F.”

The other codes to be described are based on Wheeler's 1/2 code described by Wheeler [15] and used in his early b red implementation of the Burrows–Wheeler compressor. Wheeler states that he has used this code for some years, but its provenance is otherwise unknown. The author notes that he has received many comments as "I just don't understand Wheeler's 1/2 code—however does it work?"

3.13.1 The Wheeler 1/2 Code

In a standard binary code all 0's have a weight of 0, and 1's have weights $2^0, 2^1, 2^2, 2^3, \dots$, respectively. In the Wheeler code successive 0s have weights $2^0, 2^1, 2^2, 2^3, \dots$ and successive 1's have weights $2^1, 2^2, 2^3, 2^4, \dots$. The code is terminated by any character other than 0 or 1. The first few encodings are shown in Table 3.24. The name arises because the first bit has a weight of 1 or 2, which Wheeler writes as 1/2.

Still using "/" to mean "or," the respective digit weights can be written as $(0 + 1)/(1 + 1)$, $(0 + 2)/(2 + 2)$, $(0 + 4)/(4 + 4)$, etc. Thus each weight is a standard binary weight, *plus* a constant 2^k weight for digit k . In a representation of n bits these constant weights add to $2^n - 1$; adding a further constant 1 turns these into 2^n or a single more-significant 1. Thus the Wheeler code for N is the same as the binary code for $N + 1$ with the most-significant 1 deleted, as may be seen from the last column of Table 3.24. The Wheeler 1/2 code is little more than a reinterpretation of a binary representation.

A simpler way of generating the Wheeler code for N is to encode $N + 1$ in binary and delete the most-significant or final bit. When decoding, the number is handled as binary, another 1 is implied by whatever terminates the sequence, and a 1 is subtracted.

3.13.2 Using the Wheeler 1/2 Code

Wheeler gives two ways of incorporating his code into a stream of symbols. The first is appropriate to normal text, while the second is more useful when encoding runs of 0's in the compressed output.

Table 3.24 Examples of Wheeler 1/2 Code

Bit Weights			Sums	Value N	Reverse Binary
1/2	2/4	4/8			$N + 1$
0			1	1	01
1			2	2	11
0	0		1 + 2	3	001
1	0		2 + 2	4	101
0	1		1 + 4	5	011
1	1		2 + 4	6	111
0	0	0	1 + 2 + 4	7	0001
1	0	0	2 + 2 + 4	8	1001
0	1	0	1 + 4 + 4	9	0101
1	1	0	2 + 4 + 4	10	1101
0	0	1	1 + 2 + 8	11	0011
1	0	1	2 + 2 + 8	12	1011
0	1	1	1 + 4 + 8	13	0111
1	1	1	2 + 4 + 8	14	1111

1. The binary representation of the run symbol, say x , is used to encode 0 bits in the run-length and the value $x \oplus 1$ is used for the 1 bits, emitting least-significant bits first. (\oplus represents an Exclusive-Or). Any character not in the set $\{x, x \oplus 1, x \oplus 2\}$ terminates the number (and implies the most-significant 1 bit with the simple binary decode). An $x \oplus 2$ is used as count terminator if the run is terminated by $x \oplus 1$ or $x \oplus 2$, and it is followed immediately by the terminating character.

When using this code for run compression it seems desirable to trigger it from a run of at least three identical characters. The character must occur once to establish the coding and is then confused by sequences such as “ba”. Similarly diagram triggering is confused by “eed” or “oom”; three or four characters seem to be necessary.

With a trigger of 4 symbols, encode (length-3). The character sequence “... xbbbbbbb bbb...” with a run of 11 b's might be encoded as “xbbbbcbby...”, and the sequence “... xtttttu...”, with a run of 7 t's, becomes “... xtttuuvu...”. (Experience is that this run-encoding is justified only if it gives a considerable decrease in file size.)

2. The Wheeler 1/2 code is also used to encode runs of 0's in the final coder, where the output is dominated by 0's, with many runs of 0's and negligible runs of other symbols. Encode the symbol N as $(N + 1)$ for all $N > 0$. Then *any* occurrence of a 0 or 1 indicates a run of 0's, with 0 and 1 symbols always representing the run-length in a Wheeler 1/2 code. The count is terminated by any symbol $N > 1$. While there is a slight increase in average codeword length from encoding N as $N + 1$, this encoding *never* increases the string length.

3.14 COMPARISON OF REPRESENTATIONS

There is no best variable-length coding of the integers, the choice depending on the probability distribution of the integers to be represented. A fundamental result of coding theory is that a symbol with probability P should be represented by $\log(1/P)$ bits. Equivalently a symbol represented by j bits should occur with a probability of 2^{-j} . The two simplest functions that produce a low value for large arguments n are a power function (n^{-x}) and an exponential function (e^{-n}). In both cases normalizing factors are needed to provide true probabilities, but these do not affect the basic argument.

From Table 3.1 a polynomial code represents a value n with about $x \log n$ bits. The probability P_n of an integer n should therefore be $P_n \approx n^{-x}$, showing that a γ code is well suited to power law distributions and especially where the exponent is about -2 , and the order-2 Fibonacci codes to a distribution $P_n \approx n^{-1.44}$.

A Rice code represents a value n with about $((n/2^k) + k + 1)$ bits, so that $-\log P_n \approx (n/2^k) + k + 1$. For a given k , $P_n \propto 2^{-n}$, showing that the Rice code is more suited to symbols following an exponential distribution.

But a Rice code (and by extension a Golomb code) is very well suited to peaked distributions with few small values or large values. As noted earlier, the Rice(k) code is extremely efficient for values in the general range $2^{k-1} < N < 2^{k+2}$.

Most of the other codes are true polynomial codes (γ and Fibonacci) or approximate a polynomial behavior (ω code) and are appropriate for monotonically decreasing symbol probabilities. The results for the polynomial codes are summarized in Table 3.25, giving their code lengths as functions of the corresponding binary representation. The table is based on the same range of values as Table 3.26, but with values of $\lfloor 1.1^n \rfloor$ for a good spread of values. The measured lengths are least-square fitted to the binary lengths. The theoretical expansion factor is taken from the earlier Table 3.1. Two points follow from Table 3.25:

Table 3.25 Measured Parameters of Polynomial Codes

	Expansion (x)		Bias d
	Theory	Measured	
Elias γ	2.000	2.000 ± 0	-1 ± 0
Punctured P1	1.500	$1.478 \pm .032$	$1.53 \pm .37$
Fraenkel C1	1.440	$1.436 \pm .010$	$0.50 \pm .12$
Elias ω	—	$1.386 \pm .024$	$2.61 \pm .27$
Ternary	1.262	$1.255 \pm .015$	$2.43 \pm .17$
Apostolico C1	1.137	$1.142 \pm .009$	$2.36 \pm .10$
Apostolico C2	1.137	$1.150 \pm .010$	$2.53 \pm .11$
New NF ₃	1.137	$1.131 \pm .016$	$2.50 \pm .18$

If an integer N has a binary length of ℓ_2 bits, its predicted code length is $\ell_P = x\ell_2 + d$ bits.

Table 3.26 Comparison of Representations—Codeword Lengths Over 1:2 Value Ranges

Range of Values	Elias			Ternary	Fibonacci FK ₂	Fibonacci, Order-3			Goldbach G_1	Additive Add_{256}	Best Length
	β	γ	ω			AF ₁	AF ₂	NF ₃			
1	0	<u>1</u>	<u>1</u>	4	2	3	3	3	3	2	1
2	1	<u>3</u>	<u>3</u>	4	<u>3</u>	<u>3</u>	4	4	3	4	3
3	1	<u>3</u>	<u>3</u>	6	4	5	5	5	4	4	3
4–7	2	5	6	6	<u>4.8</u>	5.8	5.8	6	5.5	5	4.8
8–15	3	7	7	7.7	<u>6.4</u>	6.9	7.3	7.1	7.3	6.8	6.4
16–31	4	9	11	8.6	<u>7.7</u>	8.1	8.4	8.1	9.1	<u>7.6</u>	7.6
32–63	5	11	12	10	<u>9.1</u>	9.3	9.6	9.3	10.9	9.9	9.1
64–127	6	13	13	11.1	<u>10.4</u>	10.4	10.6	<u>10.3</u>	12.7	11.4	10.3
128–255	7	15	14	12.3	11.9	11.5	11.8	<u>11.4</u>	14.8	13.9	11.4
256–511	8	17	16	14	13.4	<u>12.7</u>	13	<u>12.7</u>			12.7
512–1,023	9	19	17	14.9	14.7	<u>13.7</u>	14.1	13.9			13.7
1,024–2,047	10	21	18	16	16.3	<u>14.9</u>	15.1	<u>14.9</u>			14.9
2,048–4,095	11	23	19	17.8	17.6	<u>16.1</u>	16.4	<u>16.1</u>			16.1
4,096–8,191	12	25	20	18.6	19.3	<u>17.3</u>	17.6	17.4			17.3
8,192–16,383	13	27	21	20	20.6	<u>18.3</u>	18.6	<u>18.3</u>			18.3
16,384–32,767	14	29	22	21.5	22.1	19.5	19.8	<u>19.4</u>			19.4
32,768–65,535	15	31	23	22.3	23.6	20.7	20.9	<u>20.4</u>			20.4
65,536–131,071	16	33	28	24	25	<u>21.7</u>	22.1	21.9			21.7

- The observed polynomial expansions (and therefore the bases) are very close to those predicted by theory. (Even though the ω code is not strictly a polynomial it behaves as one in these measurements. The correlation between expansions is about 98% for the ω code, compared with 99% or better for the others.) In this respect the codes are just as expected.
 - It was noted earlier that the more complex and efficient codes (those with an expansion closer to 1) need more costly length specifications. This is clear for the last column, which is largely the constant overhead of the length or terminator.
- The bias is especially small for the order-2 Fibonacci code, at only half a bit. When coupled with the moderately high base, this gives good performance for small values; this code is

the best for values up to 200. Beyond 200 the order-3 Fibonacci code is shortest; its low expansion is enough to counter the overhead of the termination.

The codeword lengths of the codes are shown in Table 3.26, using the same data as underlies Table 3.25. The β code is included as a lower bound or ideal against which other codes may be compared; the γ code is, for larger values, much like an upper bound. For most codes, and except for the first three rows, each row is the average of a range of values increasing by a factor of 1.1 (but limited to different integers) within ranges 2^k to 2^{k-1} (about 7 values per “octave”). This choice of ranges exactly matches the Elias codes with a binary base, but not the ternary or Fibonacci codes whose averaged lengths thereby have a form of sampling error or uncertainty.

The Goldbach and Additive codes are handled differently. Most obviously the largest values are $N = 256$, this being the limit of the Additive code using the basis values of Table 3.23; at this value the Goldbach G_0 codeword length is starting to “run away.” Less obviously, the lengths are averaged over all values in each range. The lengths vary widely and averaging only a few leads to a form of sampling error or aliasing.

We can, however, make several observations:

1. The γ and ω codes are best for very small values, certainly to 4 and possibly to 8.
2. The order-2 Fibonacci codes are the best for larger values, to 128 and to 256 with little inefficiency. The ternary codes are nearly as good within this range.
3. For values beyond 128 the order-3 Fibonacci codes are the best codes once their inherent coding efficiency overcomes the relatively expensive termination cost. The best code alternates between the older Apostolico–Fraenkel AF_1 and the new Fibonacci code; any differences are small and probably due to the sampling error mentioned earlier. The newer code with its more complex terminator is probably preferable to the older Apostolico–Fraenkel code with its more complex generation.
4. Over much its given range the Goldbach G_1 code is similar to the Elias γ code, but distinctly inferior to the other good codes. The Additive(256) code, tuned to the range shown, is very competitive over much of its range (and is marginally best in one case) but is again starting to run away for large values.

The best general codes in these tests are the order-2 Fibonacci codes. They have a simple structure, have minimal overheads for length indication, are easy to understand, and over a wide range of values give the most compact representation or are close to the best.

3.15 FINAL REMARKS

“Good” codes are simple, with a minimum of non-numeric bits such as length indication and termination. The complexity, as well as quantity of non-numeric bits, counts against the Elias ω and higher-order Fibonacci codes. For general use the Fraenkel and Klein C^1 Fibonacci code (the FK_1 code) is a good choice, as shown by its performance for both versions of the Burrows–Wheeler compression. The Elias γ code is good for *very* skewed distributions. Rice and Golomb codes are an excellent choice where values are confined to a range of perhaps 1:10. The additive codes are useful where only small values (say $N < 250$) occur.

ACKNOWLEDGMENTS

The author thanks Dr. Robert Rice for his assistance in obtaining original documents relating to his codes.

3.16 REFERENCES

1. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ.
2. Salomon, D., 2000. *Data Compression: The Complete Reference*, 2nd Ed., Springer-Verlag, Berlin/New York.
3. Levenshtein, V. E., 1968. On the redundancy and delay of separable codes for the natural numbers. *Problems of Cybernetics*, Vol. 20, pp. 173–179.
4. Elias, P., 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, Vol. IT-21, No. 2, pp. 194–203, March 1975.
5. Even, S., and M. Rodeh, 1978. Economical encoding of commas between strings. *Communications of the ACM*, Vol. 21, No. 4, pp. 315–317, April 1978.
6. Bentley, J. L., and A. C. Yao, 1976. An almost optimal solution for unbounded searching. *Information Processing Letters*, Vol. 5, No. 3, pp. 82–87, August 1976.
7. Rice, R. F., 1979. Some practical universal noiseless coding techniques. Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena, CA, March 1979.
8. Golomb, S. W., 1966. Run-length encodings. *IEEE Transactions on Information Theory*, Vol. 12, pp. 399–401.
9. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, No. 4, pp. 490–505, April 1989.
10. Vajda, S., 1989. *Fibonacci and Lucas Numbers, and the Golden Section Theory and Applications*, Ellis Horwood, Chichester.
11. Apostolico, A., and A. S. Fraenkel, 1987. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*, Vol. IT-33, pp. 238–245.
12. Fraenkel, A. S., and S. T. Klein, 1996. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, Vol. 64, pp. 31–55.
13. Knuth, D. E., 1997. *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, 3rd ed, Addison-Wesley, Reading, MA.
14. Fenwick, P. M., 1993. Ziv–Lempel encoding with multi-bit flags. In *Proceedings of the Data Compression Conference, DCC-93*, Snowbird, UT, pp. 138–147, March 1993.
15. Wheeler, D. J., 1995. Private communication. Also “An Implementation of Block Coding,” October 1995, privately circulated.
16. Fenwick, P. M., 2002. Variable length integer codes based on the Goldbach conjecture and other additive codes. *IEEE Transactions on Information Theory*, Vol. 48, No. 8, August 2002.
17. Burrows, M., and D. J. Wheeler, 1994. A block-sorting lossless data compression algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994. Available at, <ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z>.

Huffman Coding

STEVEN PIGEON

4.1 INTRODUCTION

Codes may be characterized by how general they are with respect to the distribution of symbols they are meant to code. Universal coding techniques assume only a non-increasing distribution. Golomb coding assumes a geometric distribution [1]. Fiala and Greene's (*start*, *step*, *stop*) codes assume a piecewise uniform distribution function with each part distributed exponentially [2]. A detailed look at universal coding techniques is presented in Chapter 3. Huffman codes are more general because they assume nothing particular about the distribution, only that all probabilities are non-zero. This generality makes them suitable not only for certain classes of distributions, but for all distributions, including those where there is no obvious relation between the symbol number and its probability, as is the case with text. In text, letters go from "a" to "z" and are usually mapped onto a contiguous range, such as, for example, from 0 to 25 or from 97 to 122, as in ASCII, but there is no direct relation between the symbol's number and its frequency rank.

There are various ways of dealing with this situation. One way could be that we build a permutation function that maps the arbitrary symbol numbers to numbers corresponding to their ranks. This would effectively give us a non-increasing distribution. Once this distribution is obtained, we could use one of the aforementioned codes to do the actual coding. Since it is unlikely that the distribution obtained matches exactly one of the distributions assumed by the simpler codes, a certain inefficiency is introduced, in addition to the storage of the permutation information. This inefficiency may lead to an unacceptable loss of compression. Huffman was the first to give an exact, optimal algorithm to code symbols from an arbitrary distribution [3]. In this chapter, we will see how this algorithm manages to produce not only efficient but optimal codes. We will also see some adaptive algorithms that will change the codebook as symbol statistics are updated, adjusting themselves as probabilities change locally in the data.

4.2 HUFFMAN CODES

Huffman codes solve the problem of finding an optimal codebook for an arbitrary probability distribution of symbols. Throughout this chapter, we will use the following conventions. The uncompressed string of symbols will be the *message* and its alphabet will be the original or message alphabet. The compressed output will be composed of output symbols. We will call a *code* a string of output symbols associated with a message symbol, and a *codebook* will be the set of all codes associated with all symbols in the message alphabet. A codebook can be seen as a function F mapping symbols in A , the message alphabet, to a subset of B^+ , the set of all (non-empty) strings composed from the output alphabet B , in a non-ambiguous way.

There are, of course, many different ways of devising codebooks. However, Huffman's algorithm produces optimal codebooks, in the sense that although there may exist many equivalent codebooks, none will have a smaller average code length. Note that this is true when we respect the implicit assumption that each code is composed of an integer number of output symbol, and that each symbol receives a distinct code, which may not be true for all types of codes. Think of arithmetic coding, where the algorithm is able to assign a fractional number of output symbols to a code. Therefore, in the context of Huffman coding, “variable-length codes” really means variable-*integer*-length codes. In this section, we will be concerned with the construction of Huffman codes and their efficiency. We will also see that while we generally intend the output alphabet to be $B = \{0, 1\}$, the only requirement is that the output alphabet contains at least two symbols.

4.2.1 Shannon–Fano Coding

Shannon and Fano (see [4]) separately introduced essentially the same algorithm to compute an efficient codebook given an arbitrary distribution. This algorithm is seductively simple: First, sort all the symbols in non-increasing frequency order. Then, split this list in a way that the first part's sum of frequencies is as equal as possible to the second part's sum of frequencies. This should give you two lists where the probability of any symbol being a member of either list is as close as possible to one-half. When the split is done, prefix all the codes of the symbols in the first list with 0 and all the codes of the symbols of the second list with 1. Repeat recursively this procedure on both sublists until you get lists that contain a single symbol. At the end of this procedure, you will have a uniquely decodable codebook for the arbitrary distribution you input. It will give a codebook such as that shown in Table 4.1a.

There are problems with this algorithm. One problem is that it is not always possible to be sure (because we proceed greedily) how to split the list. Splitting the list in a way that minimizes the difference of probability between the two sublists does not always yield the optimal code! An example of a flawed codebook is given in Table 4.1b. While this algorithm is conceptually attractive because it is simple, it sometimes results in codebooks that are much worse than those given by Huffman's algorithm for the same distribution.

4.2.2 Building Huffman Codes

Huffman in his landmark 1952 paper [3] gives the procedure to build optimal variable-length codes given an arbitrary frequency distribution for a finite alphabet. The procedure is built upon a few conditions that, if satisfied, make the codebook optimal. The first conditions are as follows:

- (a) No code is a prefix of another code.
- (b) No auxiliary information is needed as delimiter between codes.

Table 4.1 Examples of Shannon–Fano Codes for Eight Symbols

(a) Symbol	Frequency	Code	(b) Symbol	Frequency	Code
a	38,416	00	a	34,225	000
b	32,761	01	b	28,224	001
c	26,896	100	c	27,889	01
d	14,400	101	d	16,900	100
e	11,881	1100	e	14,161	101
f	6,724	1101	f	4,624	110
g	4,225	1110	g	2,025	1110
h	2,705	1111	h	324	1111

Note: (a) A codebook with codes of average length 2.67. The entropy of this distribution is 2.59 bits/symbol. Discrepancy is only 0.08 bits/symbol. (b) An example of a Shannon–Fano codebook for eight symbols exhibiting the problem resulting from greedy cutting. The average code length is 2.8, while the entropy of this distribution is 2.5 bits/symbol. Here, discrepancy is 0.3 bits/symbol. This is much worse than the discrepancy of the codes shown in (a).

The first requirement implies that, for a series of bits, there is a unique way to decode it. This leaves no ambiguity as to the encoded message. The second requirement tells us that we do not need markers, or special symbols (such as the “space” symbol in the seemingly binary Morse code),¹ to mark the beginning or the end of a code.

One way to satisfy both requirements is to build a *tree-structured codebook*. If the codebook is tree-structured, then there exists a full binary tree (a binary tree is *full* when each node either is a leaf or has exactly two children) with all the symbols in the leaves. The path from the root to a leaf is the code for that leaf. The code is obtained by walking down the path from the root to the leaf and appending a 0 to the code when we go down to the left or a 1 when we go down to the right. Reaching a leaf naturally delimits the code, thus satisfying the second requirement. Were we allowed to address internal nodes, there would be codes that are prefixes of other codes, violating requirement (a). The first requirement is therefore met because no codes are assigned to internal nodes; codes designate only leaves.

Let $A = \{a_1, a_2, \dots, a_n\}$ be our alphabet for which we want a code. We rearrange the alphabet A so that symbols are listed in order of non-increasing frequency. This convention prevents us from using a permutation function $\pi(i)$ that gives the index in A of the symbol that occupies rank i . This would needlessly complicate the notation. This rearranged alphabet satisfies

$$P(a_1) \geq P(a_2) \geq \dots \geq P(a_n), \quad (4.1)$$

where $P(a)$ is a shorthand for $P(X = a)$, and X is our random source emitting the symbols. For this alphabet, we want to build an optimal set of codes with lengths

$$L(a_1) \leq L(a_2) \leq \dots \leq L(a_n). \quad (4.2)$$

It turns out that we will have to slightly modify Eq. (4.2). Let L_n be the length of the longest code. Let us suppose that there are more than two codes of length L_n that do not share prefixes of length $L_n - 1$ among themselves. Since by hypothesis we already have an optimal code, there are no other codes that are prefixes of the codes of length L_n . Since the codes of length L_n share no prefix between them and there are no other codes that are prefixes to any of the length L_n codes, it

¹ In Morse code, one uses strings of dots (·) and dashes (—) to encode letters, but the codes are separated by spaces. The duration of a dot or a space is a third of that of a dash.

means that these codes of length L_n have at least one extra bit, making them too long. Therefore, L_n should be $L_n - 1$, a contradiction! What this means is that we can drop the last bit on all these length L_n codes. This in turn implies that at least two of the longest codes of length L_{max} must share a prefix of length $L_{max} - 1$. The same kind of argument will prevent an algorithm that builds an optimal codebook from giving extra, useless bits to codes. Equation (4.2) becomes

$$L(a_1) \leq L(a_2) \leq \cdots \leq L(a_{n-1}) = L(a_n), \quad (4.3)$$

where the last \leq is now $=$. We now have the following conditions to satisfy:

- (a) No code is a prefix of another code.
- (b) No auxiliary information is needed as delimiter between codes.
- (c) $L(a_1) \leq L(a_2) \leq \cdots \leq L(a_{n-1}) = L(a_n)$.
- (d) Exactly two of the codes are of length L_{max} and are identical except for their last bit.
- (e) Every possible code of lengths $L_{max} - 1$ is either already used or has one of its prefixes used as a code.

Surprisingly enough, these requirements will allow a simple algorithm to fulfill them. The key requirement is requirement (c). While Huffman codes generalize to any number of coding digits greater than 1, as we will see in Section 4.2.3, let us restrict ourselves for now to the case of a binary output alphabet, the usual {0, 1}.

The algorithm will proceed iteratively. The process is illustrated in Fig. 4.1. At the start, all symbols are given a tree node that is the root of its own subtree. Besides the symbol and its probability, the node contains pointers to a right and a left child. They are initialized to null,

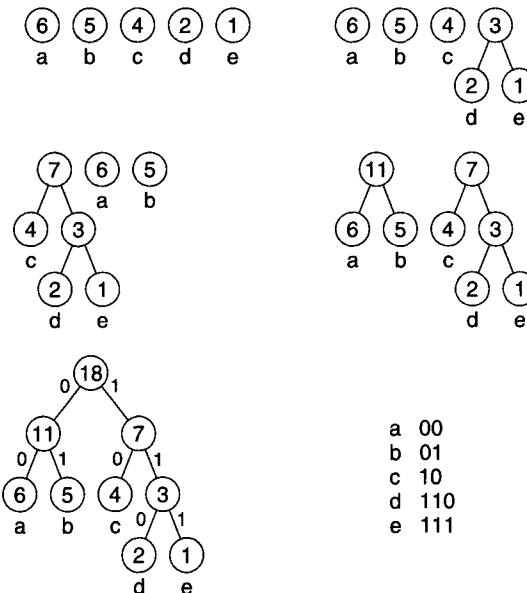


FIGURE 4.1

How Huffman's algorithm proceeds to build a code. At the beginning, all nodes are roots of their own degenerate tree of only one leaf. The algorithm merges the trees with the least probability first (note that in the figure, the *frequencies* are shown) and repeats this procedure until only one tree is left. The resulting code is $\{a \rightarrow 00, b \rightarrow 01, c \rightarrow 10, d \rightarrow 110, e \rightarrow 111\}$, for an average length of 2.17 (while the entropy is 2.11 bits/symbol, a discrepancy of 0.06 bits/symbol).

symbolized here by \perp . All the roots are put in a list L . Requirement (c) asks for the two symbols with lowest probability to have codes of the same length. We remove the two roots having the smallest probabilities from L ; let them be a and b . We create a new root c having probability $P(a) + P(b)$ and having children a and b . We add c to L . This will cause a and b to share a common prefix, the code for c . We find ourselves with one less tree in L . If we repeat this until only one tree is left in L , we will have completed the tree-structured code satisfying *all* the requirements. The algorithm in pseudo-code looks like this:

```

 $L = \{(a_1, P(a_1), \perp, \perp), (a_2, P(a_2), \perp, \perp), \dots, (a_n, P(a_n), \perp, \perp)\}$ 
While  $|L| > 1$ 
{
     $a = \min_P L$ 
     $L = L - \{a\}$ 
     $b = \min_P L$ 
     $L = L - \{b\}$ 
     $c = (\perp, P(a) + P(b), b, a)$ 
     $L = L \cup \{c\}$ 
}.

```

From there, one has the tree that describes the codebook, but the codes per se are not assigned yet. We know, however, how to assign the codes, as we already have described the procedure earlier. The codes are obtained by walking down the path from the root to the leaves and appending a 0 if we go down to the left or a 1 if we go down to the right.² Reaching a leaf naturally determines the end of the code. We only have to copy the code that has been accumulated as a bit string in an array indexed by the symbol in the leaf we reached.

The encoding process is straightforward. We emit the bit string contained in the table, at the address indexed by the symbol. Decoding is just a bit more complicated. Since we do not know beforehand the length of the code that we are about to read, we will have to walk the tree as we read bits one by one until we reach a leaf which will correspond to the decoded symbol. This general procedure can be improved in many ways, as we will see in Section 4.5. We will also see in a subsequent section that not only is the procedure given by Huffman simple, it also generates optimally efficient codes.

4.2.3 N -ary Huffman Codes

Although binary coding is the most prevalent form of coding, there are cases where more than two coding symbols are preferable. In the context of electromechanical devices, one often has a channel that allows for more than two voltage values: It would be a waste not to use this possibility to our advantage! Communications can be made a lot more efficient if we have $m \gg 2$ coding symbols at our disposal. The first thing that comes to mind is simply to generalize requirement (c) by picking not only the two nodes with the smallest probability but the m nodes with the smallest probability. While this seems like a good idea, we soon encounter a problem. Let us say that we have $m = 3$ coding symbols and a source alphabet size of $n = 6$. We apply the algorithm as we described earlier, but rather than picking the two symbols with the smallest probability of occurrence, we pick three. After the second iteration, we find ourselves with only two nodes left! We cannot just pick the three less probable subtrees.

² Note that this is arbitrary. We could as well choose a 1 when going down to the left and a 0 when going down to the right.

Fortunately, there is a rather simple fix. The first time, instead of picking m nodes, pick $2 \leq m' \leq m$ nodes. In order to make sure that all subsequent steps merge m nodes, we first compute $a = n \bmod (m - 1)$ and then find $m' \equiv a \pmod{m - 1}$. As $2 \leq m' \leq m$, we test the different values of m' until we find one that satisfies $m' \equiv a \pmod{m - 1}$.

For example, for $n = 6, m = 3$ we find $m' = 2$. For $n = 7, m = 3, m' = 3$. For $n = 8, m = 3$, we find $m' = 2$ again. The generalized algorithm will be based on generalized requirements (c) and (d), which now read

- (c') At least two, and not more than m , codes will be of length L_{max} .
- (d') At least two, and at most m , codes of length L_{max} are identical except for the last m -ary digit.

4.2.4 Canonical Huffman Coding

The reader may have noticed that the codes assigned to symbols by Huffman's algorithm described in Section 4.2.2 are in numerical order. It generates a code such that if $P(a_i) \leq P(a_j)$, then the code for a_i numerically precedes the code for a_j . The algorithm systematically assigns the right child to the subtree of least probability and the left child to the other. If we use the convention that going down to the left means 0 and down to the right means 1, we get a *canonical codebook*, where $code(a_i) < code(a_j)$ iff $P(a_i) \leq P(a_j)$.

The use of canonical codes is not self-evident, but it turns out that fast decoding algorithms, as we will see in Section 4.5, can exploit this property. If the codes are assigned somewhat randomly, there is no easy way to know the length of the current code before the end of decoding. For example, take the code obtained in Fig. 4.1. From the code $\{00, 01, 10, 110, 111\}$, we can see that we can read 2 bits to find out if the code represents a, b , or c or if we must read another bit to find whether it is d or e . We cut the number of bit extractions and comparison by $\frac{1}{3}$. This may not seem that spectacular because in fact we skipped only 1 bit, but in larger codebooks, we tend to get larger numbers of codes of the same length, thus leading to possibly greater savings in decoding time.

4.2.5 Performance of Huffman Codes

Let us now derive the bounds on the efficiency of Huffman codes. We will first consider the average code length of Huffman codes for some source X ,

$$L(X) = \sum_{a_i \in A} P(a_i)L(a_i), \quad (4.4)$$

with $P(a_i)$ being shorthand for $P(X = a_i)$. We know, since the codes are uniquely decodable, that they must satisfy the Kraft–McMillan inequality; that is,

$$\sum_{a_i \in A} 2^{-L(a_i)} \leq 1. \quad (4.5)$$

Knowing this, we want to show that

$$\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + 1, \quad (4.6)$$

where $\mathcal{H}(X)$ is the entropy for source X . The entropy of a random source X is defined as

$$\mathcal{H}(X) = - \sum_{a_i \in A} P(a_i) \lg P(a_i),$$

where $\lg(x)$ is $\log_2(x)$. We will prove the bounds in two parts. The first part will be that, if the codes are uniquely decodable, $L(X) \geq \mathcal{H}(X)$. We will then prove that given a Huffman codebook, $L(X) < \mathcal{H}(X) + 1$.

The difference between entropy and average code length is

$$\begin{aligned}
 \mathcal{H}(X) - L(X) &= - \sum_{a_i \in A} P(a_i) \lg P(a_i) - \sum_{a_i \in A} P(a_i)L(a_i) \\
 &= \sum_{a_i \in A} P(a_i)(-\lg P(a_i) - L(a_i)) \\
 &= \sum_{a_i \in A} P(a_i)(-\lg P(a_i) + \lg 2^{-L(a_i)}) \\
 &= \sum_{a_i \in A} P(a_i) \lg \frac{2^{-L(a_i)}}{P(a_i)} \\
 &\leq \lg \left(\sum_{a_i \in A} 2^{-L(a_i)} \right) \leq 0.
 \end{aligned} \tag{4.7}$$

This last inequality is an application of Jensen's inequality. It states that if a function is concave (convex down, often written convex \cap), then $E[f(X)] \leq f(E[X])$. $E[X]$ is the expectation of random variable X , $\lg(\cdot)$ is a convex \cap function, and the part of Eq. (4.7) before the \leq part is exactly that: the expectation of the number of bits wasted by the code. If $\mathcal{H}(X) - L(X) \leq 0$, it means that $L(X) \geq \mathcal{H}(X)$. The length is therefore never shorter than the entropy, as one could have guessed from the fact that a Huffman codebook satisfies the Kraft–McMillan inequality.

To show that the upper bound holds, we will show that an optimal code is always less than 1 bit longer than the entropy. We would like to have codes that are exactly $-\lg P(a_i)$ bits long, but this would ask for codes to have any length, not only integer lengths. The Huffman procedure obviously produces codes that have an integer number of bits. Indeed, the length of the code for symbol a_i is

$$L(a_i) = \lceil -\lg P(a_i) \rceil,$$

where $\lceil x \rceil$ is the ceiling of x , the smallest integer greater than or equal to x . We can express $\lceil x \rceil$ as $\lceil x \rceil = x + \varepsilon(x)$. We have

$$-\lg P(a_i) \leq L(a_i) < 1 - \lg P(a_i). \tag{4.8}$$

The strict inequality holds because $\lceil x \rceil - x = \varepsilon(x) < 1$. It also means that $2^{-L(a_i)} \leq P(a_i)$, which gives

$$\sum_{a_i \in A} 2^{-L(a_i)} \leq \sum_{a_i \in A} P(a_i) = 1.$$

This means that a codebook with code lengths $L(a_i)$ satisfies the Kraft–McMillan inequality. This implies that *there must exist* a code with these lengths! Using the right inequality in Eq. (4.8), we finally get

$$L(X) = \sum_{a_i \in A} P(a_i)L(a_i) < \sum_{a_i \in A} P(a_i)(-\lg P(a_i) + 1) = \mathcal{H}(X) + 1,$$

which concludes the proof on the bounds $\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + 1$. This means that a codebook generated by Huffman's procedure always has an average code length less than 1 bit longer than what entropy would ask for. This is not as tight a bound as it may seem. There are proofs, assuming something about the distribution, that give much tighter bounds on code lengths. For example, Gallager [5] shows that if the distribution is dominated by $p_{max} = \max_{a_i \in A} P(a_i)$, the expected code length is upper-bounded by $\mathcal{H}(X) + p_{max} + \sigma$, where $\sigma = 1 - \lg e + \lg \lg e \approx 0.0860713320 \dots$. This result is obtained by the analysis of internal node probabilities defined

as the sum of the probabilities of the leaves it supports. Capocelli *et al.* [6] give bounds for a code where $\frac{2}{9} \leq p_{max} \leq \frac{4}{10}$ with no other information on the other p_i , as well as bounds for $\frac{1}{3} \leq p_{max} \leq \frac{4}{10}$ and for distributions for which only p_{max} and p_{min} are known. Buro [7], under the assumption that $P(a_i) \neq 0$ for all symbols and that Eq. (4.1) holds, characterizes the maximum expected length in terms of ϕ , the golden ratio!

Digression 4.1. Although we have shown that the average length of Huffman codes are generally quite close to the entropy of a given random source X , we have not seen how long an individual code can get. We can see that an alphabet with $P(a_i) = 2^{-\min(i, n-1)}$ will produce a code of length $L(a_i) = \min(i, n-1)$. Campos [8] pointed out another way of producing maximally long codes. Suppose that for n symbols, we have the probability function

$$P(X = a_i) = \frac{F_i}{F_{n+2} - 1},$$

where F_i is the i th Fibonacci number. The sum of the first n Fibonacci numbers is

$$\sum_{i=1}^n F_i = F_{n+2} - 1.$$

This set of probabilities also generates a Huffman code with a maximum code length of $n - 1$. To avoid ludicrously long codes, one could consider limiting the maximum length of the codes generated by Huffman's algorithm. There are indeed papers where such algorithms are presented, as we will see in Section 4.3.4.

Katona and Nemetz [9] show that using Fibonacci-like probabilities causes the Huffman algorithm to produce not only maximally long codes but also the largest possible differences between assigned code lengths and what would be strictly needed according to the entropy. They use the notion of self-information, $I(a)$, defined as $I(a) = -\lg P(X = a)$, to show that the ratio of the produced code length to the self-information is $1/\lg \phi = 1.44 \dots$. Here, $\phi = \frac{1}{2}(1 + \sqrt{5})$.

Digression 4.2. We noted earlier that the choice of assigning a 0 when going down to the left and a 1 when going down to the right in the tree was arbitrary but led to canonical codebooks. If we did not care about the canonicity of the codebook, we could as well reverse the use of 0 and 1. This leads to the observation that for a given tree, multiple codebooks are possible. Suppose, for example, that each time a branching occurs, we decide randomly on which branch is designated by 0 or 1. This would still result in a valid Huffman code tree. Recall that a Huffman tree is *full*, and this property ensures that if there are n leaves (one for each of the symbols in the alphabet), then there are $n - 1$ internal nodes. Since each internal node has the choice on how to assign 0 or 1 to its children, it is the same as saying that each internal node has two possible states: $(0 \rightarrow \text{left}, 1 \rightarrow \text{right})$ or $(0 \rightarrow \text{right}, 1 \rightarrow \text{left})$. Because each of the $n - 1$ nodes has two possible states, there are 2^{n-1} possible different equivalent Huffman code trees, all optimal.

4.3 VARIATIONS ON A THEME

Because Huffman codebooks are very efficient, it is not surprising that they would appear in a number of places and in numerous guises. They do not, however, satisfy everybody as is. Some will need very large sets of symbols, and others will want them to be length constrained. There are also cases where basic Huffman codes are not as efficient as we may hope. In this section, we will consider a few common modifications to Huffman's algorithm to produce slightly different codes.

We will get a look at modified Huffman codes, Huffman prefixed codes, and extended Huffman codes.

4.3.1 Modified Huffman Codes

In facsimile (a.k.a. fax) transmission, the document to be transmitted is scanned into lines of 1728 pixels, for up to 3912 lines. Each line of the document image is translated into runs of white and black pixels, which are coded, in accordance with CCITT Group 3 recommendations, by *modified Huffman* codes. This modified Huffman scheme allocates two sets of codes, one for the white runs and one for the black runs. The reason for this is that the distribution of the number of successive white pixels in a scanned text document differs substantially from the distribution of the number of successive black pixels since there are usually black pixels only where there is text. Using a unique codebook for both distributions would be inefficient in terms of compression.

The codes are “modified Huffman” codes because instead of allocating a code for each possible run length $0 \leq l < 1728$, codes are allocated for all $0 \leq l < 63$, then for every l that is a multiple of 64, up to 2560. The codes for $0 \leq l < 63$ are termed Termination Codes because they always encode the number of pixels at the end of a run, while the codes for the runs of lengths that are multiples of 64 are termed Make-Up Codes, since they encode the body of the runs. The reader interested in the intricacies of facsimile coding is referred to the chapter on Facsimile Compression.

4.3.2 Huffman Prefixed Codes

Another way to devise a codebook for a very large number of symbols is to use probability classes and to compute the codebook for the classes only. For example, let us pretend that we have an alphabet A with a very large number of symbols, something like 2^{32} . We can partition A into equivalence classes in a way such that all symbols that have roughly the same probability end up in the same equivalence class. The idea is to create a codebook in two parts. The first part, or prefix, is a Huffman code for equivalence classes, and the second part, or suffix, is simply an index within a given equivalence class to the desired symbol.

Let us define the equivalence classes and the concept of “having roughly the same probability as” more rigorously. Two symbols a and b , will be equivalent $a \equiv_p b$ iff $\lceil -\lg P(a) \rceil = \lceil -\lg P(b) \rceil$. Let C_{p_i} be the equivalence class for probability p_i . All $a \equiv_{p_i} b$ are in class C_{p_i} . Instead of computing individual codes for each of the symbols $a_i \in A$, using $P(a_i)$, we will build the codes for the classes C_{p_i} with probabilities $P(C_{p_i}) = \sum_{a \in C_{p_i}} P(a)$.

The resulting prefix codebook for the equivalence classes satisfies Eq. (4.6). This means that the codes for the prefix part will always be less than 1 bit longer than the entropy. The suffix part is only the index of a_i into C_{p_i} . Using a natural code for the index within an equivalence class, $\lceil |C_{p_i}| \rceil$ bits suffice. The resulting code is therefore always less than 2 bits longer than the entropy. If we use a *phase-in* code [10], the length suffix of the suffix code is essentially $\lg |C_{p_i}|$ when $|C_{p_i}|$ is large enough and given that symbols within an equivalence class are chosen according to a uniform random variable. Table 4.2 shows such a code.

4.3.3 Extended Huffman Codes

If the source alphabet is rather large, p_{max} is likely to be comparatively small. On the other hand, if the source alphabet contains only a few symbols, the chances are that p_{max} is quite large compared to the other probabilities. Recall that we saw in Section 4.2.5 that the average code length is upper-bounded by $H(X) + p_{max} + 0.086$. This seems to be the assurance that the code is never very bad.

Table 4.2 An Example of a Large Number of Symbols Represented by a Small Number of Probability Classes, and the Resulting Huffman Prefix Code

Probability	Prefix	Suffix Bits	Range
$P(0 \leq X < 4) = \frac{1}{2}$	0	xx	0–3
$P(4 \leq X < 11) = \frac{1}{4}$	10	xxx	4–11
$P(12 \leq X < 27) = \frac{1}{16}$	1100	xxxx	12–27
$P(28 \leq X < 59) = \frac{1}{16}$	1101	xxxxx	28–59
$P(60 \leq X < 128) = \frac{1}{16}$	1110	xxxxxx	60–123
$P(124 \leq X < 251) = \frac{1}{16}$	1111	xxxxxxx	124–251

It is unfortunately not so. Consider the following example. Say that source alphabet is $A = \{0, 1\}$ and that output alphabet $B = \{0, 1\}$. Regardless of the probabilities, the average code length is still $\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + p_{max} + 0.086 \dots$, but *absolutely no compression is achieved*, since the resulting Huffman codebook will be $\{0 \rightarrow 0, 1 \rightarrow 1\}$.

How do we get out of this tar pit? One solution is to use extended alphabets. The extended alphabet of order m generated from A is given by

$$A^m = \underbrace{A \times A \times \cdots \times A}_{m \text{ times}} = \{a_1 a_1 \dots a_1 a_1, \underbrace{a_1 a_1 \dots a_1 a_2, \dots, a_n a_n \dots a_n}_{m \text{ symbols}}, \dots\},$$

where $A \times A$ is to be interpreted as the Cartesian product of A with itself. The probability of a symbol a_i^m of A^m is simply (if one assumes that the original symbols of A are independent and identically distributed) given by

$$P(a_i^m) = \prod_{j=1}^m P(a_{ij}^m).$$

This results in n^m symbols and in the same number of probabilities. The resulting Huffman codebook satisfies, under the independent identically distributed (i.i.d.) hypothesis,

$$\mathcal{H}(X^m) \leq L(X^m) < \mathcal{H}(X^m) + 1$$

and that gives, for the original symbols of A , an average code length

$$\frac{1}{m} \mathcal{H}(X^m) \leq \frac{1}{m} L(X^m) < \frac{1}{m} (\mathcal{H}(X^m) + 1)$$

or, after simplification,

$$\mathcal{H}(X) \leq L(X) < \mathcal{H}(X) + \frac{1}{m}.$$

Therefore, we can correctly conclude that packing symbols lets us get as close as we want to the entropy, as long as we are willing to have a large m . The problem with a large m is that the number of symbols (and probabilities) for alphabet size n is n^m , which gets unmanageably large except for trivially small values of both n and m .

In such a situation, other coding techniques may be preferable to Huffman coding. For example, we could build a k -bit Tunstall code [11]. A Tunstall code associates variable-length strings of symbols to a set of fixed-length codes rather than associating a code to every symbol. We can build a codebook with the $2^k - n$ first codes associated with the $2^k - n$ most *a priori* likely strings, under i.i.d assumptions, and keep n codes for literals, because not all possible strings will

be present in the the dictionary. We could also consider using another algorithm altogether, one that naturally extracts repeated strings from the input. One such algorithm is Welch's variation [12] on the Ziv–Lempel dictionary-based scheme [13], known as LZW. We could also consider the use of arithmetic coding and estimate the probabilities with a small-order probabilistic model, like an order m Markov chain or a similar mechanism.

4.3.4 Length-Constrained Huffman Codes

One may want to limit in length the codes generated by Huffman's procedure. They are many possible reasons for doing so. One possible reason for limiting the length of Huffman-type codes is to prevent getting very long codes when symbol probabilities are underestimated. Often, we get approximated probability information by sampling part of the data. For rarer symbols, this may lead to symbols getting a near zero probability, despite their occurring much more frequently in the data. Another reason could be that we want to limit the number of bits required to be held in memory (or CPU register) in order to decode the next symbol from a compressed data stream. We may also wish to put an upper bound on the number of steps needed to decode a symbol. This situation arises when a compression application is severely constrained in time, for example, in multimedia or telecommunication applications, where timing is crucial.

There are several algorithms to compute length-constrained Huffman codes. Fraenkel and Klein [14] introduced an exact algorithm requiring $O(n \lg n)$ steps and $O(n)$ memory locations to compute the length-constrained code for a size n alphabet. Subsequently, Milidiú *et al.* [15] presented an approximated fast algorithm asking for $O(n)$ time and memory locations, which, much like Moffat's and Katajainen's algorithm [16], computes the code lengths *in situ* using a dynamic programming-type algorithm. Finally, Larmore and Hirschberg [17] introduced another algorithm to compute length-constrained Huffman codes. This last paper is extensive as it describes both mathematical background and implementation details.

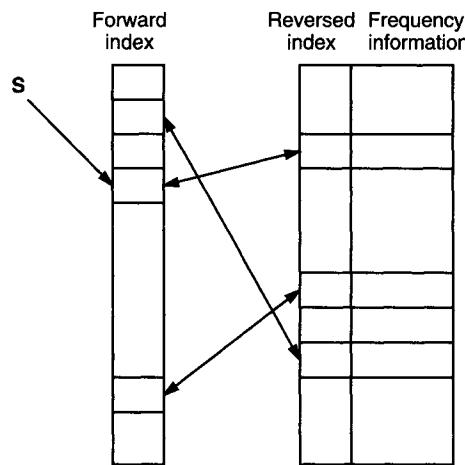
4.4 ADAPTIVE HUFFMAN CODING

The code generated by the basic Huffman coding algorithm is called the *static Huffman Code*. The static Huffman code is generated in two steps. The first step is the gathering of probability information from the data. This may be done either by exhaustively scanning the data or by sampling some of it. If we opt for exhaustively scanning the data, we may have to deal with large quantities of data, if it is even possible to scan the complete data set at all. In telecommunications, “all the data” does not mean much because the data may not yet be available as it arrives in small packets from a communication channel. If we opt for sampling some of the data, we expose ourselves to the possibility of very bad estimates of the probabilities, yielding an inefficient code.

Fortunately, there are a number of ways to deal with this problem. The solutions bear the common name *dynamic* or *adaptive Huffman coding*. In dynamic Huffman coding, we will update the codes as we get better estimates of the probability, either locally or globally. In this section, we will get a look at the various schemes that have been proposed to achieve adaptive compression.

4.4.1 Brute Force Adaptive Huffman

One naive way to do adaptive Huffman coding is the brute force approach. One could recompute the tree each time the probabilities are updated. As even an efficient implementation (such as, for example, Moffat's and Katajainen's astute method [16], Section 4.5) asks for at least $O(n)$ steps, for alphabet size n , it seems impractical to update the codes after each symbol.

**FIGURE 4.2**

A simple structure to keep symbol rank information. The forward index uses the symbol directly to find where in the rank table the frequency information is stored. The reversed index is used to update the forward index when the symbol rank changes up or down.

What if we compute the new codes every now and then? One could easily imagine a setting where we recompute the codes only after every k symbols have been observed. If we update the codes only every 1000 symbols, we divide the cost of this algorithm by 1000, but we pay in terms of compression efficiency. Another option would be to compute the code only when we determine that the codes are significantly out of balance. Such would be the case when a symbol's rank in terms of frequency of occurrence changes.

This is most easily done with a simple ranking data structure, described in Fig. 4.2. A forward index uses the symbol as the key and points at an entry in the table of frequencies. Initially, all the frequencies are set to 1 (zero could lead to problems). Each time a symbol is observed, we update its frequency count, stored in the memory location pointed to by the forward index. The frequency is then bubbled up so as to maintain the array sorted in non-increasing frequency order. As in bubble sort, we swap the current frequency count with the previous frequency count while it is larger. We also update the forward index by swapping the forward index pointers (we know where they are by the reversed index). This procedure is expected $O(1)$ amortized times since it is unlikely, at least after a good number of observations, that a symbol must be bubbled up from the bottom of the array to the top. However, it can be $O(n)$ in the beginning when all symbols have a frequency count of 1. After, and if, a swap occurred in the table we recompute the codes using any implementation of Huffman's procedure. Experimental evidence shows that this algorithms performs slightly better than Vitter's algorithm (which we will see in Section 4.4.3) and that the number of tree recomputations is low for text files. Experiments on the Calgary Corpus text files show that this algorithm recomputes the tree for about 2–5% of the symbols, giving an acceptable run time even for large files.

This algorithm is not quite as sophisticated as the algorithms we will present in the next few subsections. Indeed, it does a lot more work than seems necessary. With this simple algorithm, we recalculate the whole codebook even when only a few symbols exchange rank. Moreover, exchanging rank does not mean that the symbols also have new code lengths or even that they must change codes.

4.4.2 The Faller, Gallager, and Knuth (FGK) Algorithm

Faller [18] and Gallager [5] independently introduced essentially the same algorithm, a few years apart. It was later improved upon by Cormack and Horspool [19]. The same improvements appeared again 1 year later, this time proposed by Knuth [20]. The adaptive algorithm proposed by Gallager requires some introductory material before it is introduced.

A tree has the *sibling property* if every internal node other than the root has a sibling and if all nodes can be enumerated in nondecreasing probability order. In such a tree, there are exactly $2n - 2$ nodes. A useful theorem introduced by Gallager states that a full tree is a Huffman tree if, and only if, it has the sibling property. Let us define the depth of a node as the number of edges that separate it from the root. The root has a depth of 0. Internal node probabilities are defined as the sum of the probabilities of the leaves that they support.

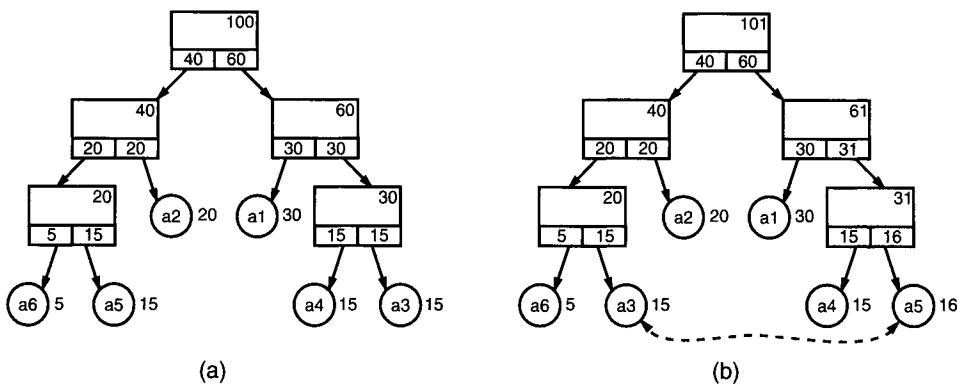
We will maintain a tree such that all nodes at a given depth d have a probability that is less than or equal to all the probabilities of nodes at depth $d - 1$ and above. A tree that has this property for all depths is said to be ordered if the 0-bit goes to the most likely of the two children for each node. Furthermore, a tree is said to be *lexicographically ordered* if, for any depth $d \geq 1$, all the nodes at depth d have smaller probabilities than nodes at depth $d - 1$, and all nodes at depth d can be enumerated in non-decreasing order, from left to right, in the tree. Finally, a binary prefix code tree is a Huffman tree if, and only if, it is lexicographically ordered.

The update procedure proposed by Gallager requires that the tree remain lexicographically ordered at all times. We will not discuss how the data structures are used to maintain efficient access to data within the tree, as Knuth's paper [20] describes in a complete way a very efficient set of data structures and subalgorithms to update the tree.

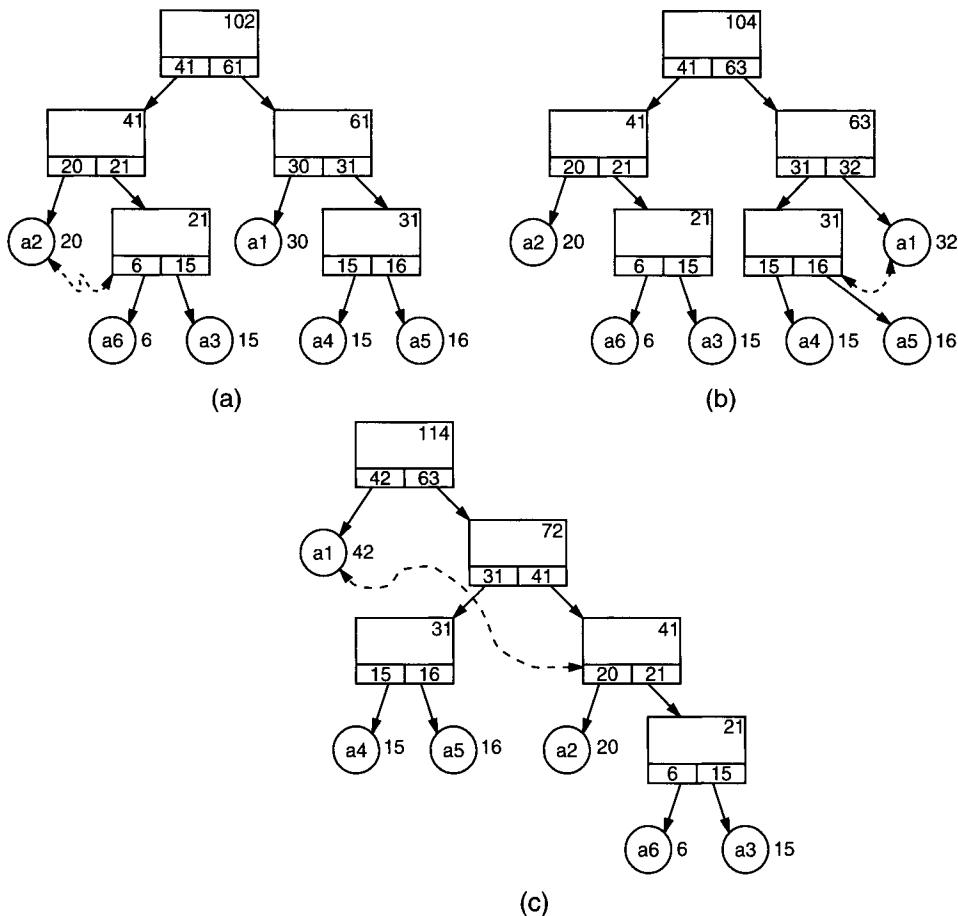
When a new symbol is observed, its old code (here also obtained by walking down the tree from the root to the leaf that contains this symbol, exactly as in the basic Huffman algorithm) is transmitted, so that the decoder remains synchronized with the encoder. After the code is emitted, the update procedure begins. The frequency count associated with the leaf is incremented by 1. Let us say that this leaf, a , is at depth d . After its counts have been incremented, we check if all the nodes at depth d are still in nondecreasing order. If not, we exchange this leaf with the rightmost node with a count lower than a 's count. Let this node be b . Node b does not have to be at depth d ; it can be at depth $d - 1$ or above. After the exchange, we update b 's parent frequency count recursively up to the root. Since b 's frequency is less than or equal to a 's frequency before the update, no other exchange will be provoked by the recursive frequency updates. On the other hand, while we recursively update a 's ancestors up to the root, we may have to do more exchanges. Figures 4.3 and 4.4 illustrate the process. The initial state of the tree is adapted from the example given by Gallager in [5].

We have seen that this algorithm takes only positive unit increments on the frequency count and that it does not deal explicitly with symbols with zero frequency. The problem with only being able to increment the frequency is that the algorithm captures no local trend in the data; it only converges slowly on the long-term characteristics of the distribution. Gallager proposes a “frequency decay” method where, every N symbols, all frequencies are multiplied by some constant $0 < \alpha \leq 1$ to keep them relatively small so that an increment of 1 has a good chance to provoke an update in the tree. There are problems with this method, such as, do we round or truncate the result after multiplication with α ? Another problem is how to choose N and α in an adaptive way.

Cormack and Horspool proposed an algorithm to update the frequency by any positive amount. This procedure was generalized by Knuth to allow negative integers as well. The advantage with negative integer increments is that we can keep a size M window over the source to compress, incrementing the frequencies according to the symbols that are coming in and decrementing them according to the symbols that are leaving the window. This method provides a better mechanism to

**FIGURE 4.3**

A single update. (a) The tree before the update; (b) the tree after the observation of symbol a_5 . The dotted arrow shows where the exchanges occurred.

**FIGURE 4.4**

After multiple updates. (a) The tree after the observation of symbol a_6 . (b) The tree after 2 observations of symbol a_1 . At this point, symbol a_1 is as far right as it can get on this level, at depth 2. If it is updated again, it will climb to the previous level, depth 1. (c) The updated tree after 10 observations of a_1 . Note that a complete subtree has been exchanged, as is shown by the dotted arrow.

capture non-stationarity in the source. The hypothesis of stationarity may make sense for certain sources, but not for all. There are many cases where statistics clearly change according to the source's history.

Gallager's algorithm starts with all symbols as leaves and the tree is originally *complete*. A tree is said to be complete if all leaves lie on at most two different depths. If n , the alphabet size, is of the form $n = 2^k$, then all leaves are at depth k . Knuth further modified Gallager's algorithm to include an escape code used to introduce symbols that have not been observed yet. Since it is possible that for a given source, only a subset of the symbols will be generated for a given message, it is wasteful to allocate the complete tree as soon as compression starts. Knuth's idea is the following. The tree starts as a single leaf, the special zero-frequency symbol. This leaf will be the special escape code. When a new symbol is met, we first emit the code for the special leaf and then emit the new symbol's number among the remaining unobserved symbols, from 0 to $n - r - 1$, if r symbols have been observed so far. This asks for $\lceil \lg(n - r) \rceil$ bits. We create the new symbol's leaf as the sibling of the zero-frequency leaf, creating a parent where the zero-frequency leaf was. This maintains exactly 1 zero-frequency leaf at all times. When the last unobserved symbol is encountered, the special zero-frequency leaf is simply replaced by this symbol, and the algorithm continues as Gallager described. The obvious advantage of using this technique is that the codes are very short right from the start.

Knuth gives the complexity of the update and “downdate” (negative increment) procedures as being $O(-\lg P(X = a_i))$ given we observe symbol a_i . This gives a total encoding complexity, for a sequence of length m , of $O(m\mathcal{H}(X))$, since the expected value of $-\lg P(X = a_i)$ is $\mathcal{H}(X)$. This algorithm is suitable for adaptive Huffman coding even for low-computational-power environments. The memory requirement is $O((2n - 1)c)$, where c is the cost (in bytes or bits) of storing the information of a node in memory.

4.4.3 Vitter's Algorithm: Algorithm Λ

Vitter's algorithm Λ [21, 22] is a variation of algorithm FGK. The main difference between algorithm FGK and Vitter's algorithm Λ is the way the nodes are numbered. Algorithm Λ orders the nodes the same way as algorithm FGK, but also asks for leaves of a given weight to always precede internal nodes of the same weight and depth, while in algorithm FGK it was sufficient to order weights of nodes at a given depth from left to right, with no distinction of node type. This modification (as shown in [21]) is enough to guarantee that algorithm Λ encodes a sequence of length s with no more than s bits more than with the static Huffman algorithm, without the expense of transmitting the probability (or code) table to the decoder. In [22], Vitter gives a complete pseudo-Pascal implementation of his algorithm, where most of the data structures introduced in [20] are present. The code is clear enough that we should not have any major problem translating it to our favorite programming language.

4.4.4 Other Adaptive Huffman Coding Algorithms

Jones introduced an algorithm using splay trees [23]. A splay tree is a data structure where recently accessed nodes are promoted to a position near the root of the tree [24]. The original splay tree is in fact an ordinary search tree, with information in every node, and not necessarily full. In the case of a Huffman-type code tree, the symbols are only in the leaves, and the internal nodes contain no symbol information. Jones presents an algorithm that modifies the splay tree algorithm of Tarjan enough so that splaying maintains a valid Huffman tree. Splaying is also reduced. Instead of promoting a symbol directly to be one of the root's children, a leaf is allowed to climb only two

levels at a time, and depth is reduced by a factor of 2. This algorithm performs significantly worse than the other adaptive Huffman algorithms in most cases. In some cases the message contains long series of repeating symbols. In these cases the repeated symbol climbs rapidly and quickly gets the shortest code, thus achieving potentially better compression than the other algorithms on run-rich data. The splaying provides Jones' algorithm with what we may call short-term memory.

Pigeon and Bengio introduced algorithm M where splaying is controlled by a weight rule [25–27]. A node splays up one level only if it is as heavy as its parent's sibling. While algorithm Λ almost always beats algorithm M for small alphabets ($n = 256$), its results are within $\approx 5\%$ of those of algorithm Λ on the Calgary Corpus. The situation is reversed when we consider very large alphabets, say $n = 2^{16}$: Algorithm M beats algorithm Λ by $\approx 5\%$ on the Calgary Corpus files treated as being composed of 16-bit symbols. This suggests the possible usefulness of algorithm M in the compression of Unicode files. Another original feature of algorithm M is that not all symbols get a leaf. The leaves represent *sets* of symbols. A set is composed of all the symbols of the same frequency. The codes are therefore given to frequency classes rather than to individual symbols. The complete code is composed of a prefix, the Huffman-like code of the frequency class, and of a suffix, the number of the symbol within this frequency class. The suffix is simply the natural code for the index. The fact that many symbols share the same leaf also cuts down on memory usage. The algorithm produces codes that are never more than 2 bits longer than the entropy, although in the average case, the average length is very close to the average length produced by the static Huffman algorithm.

4.4.5 An Observation on Adaptive Algorithms

McIntyre and Pechura [28] advocate the use of static Huffman coding for small files or short strings, since the adaptation process needs a relatively large number of symbols to be efficient. Furthermore, since the number of observations for each symbol is small, it is unlikely that an adaptive algorithm can produce an efficient codebook. This situation arises when the number of symbols to be encoded is large but still relatively small compared to the alphabet size. Suppose we want an adaptive code for integers smaller than 2^{16} . One would expect to have to observe many times the number of symbols in the alphabet before getting a good code. If, on the contrary, the number of symbols to be encoded is small, say a few tens, then this code will do much worse than a static, preoptimized, code. We will then have to consider using an alternative, such as an adaptive Huffman prefix code, much like that discussed in Section 4.3.2, but where the codes for the equivalence classes are updated by an adaptive algorithm such as algorithm M .

4.5 EFFICIENT IMPLEMENTATIONS

The work on efficient implementations of Huffman coder/decoder pairs, or codecs, takes essentially two directions, which are almost mutually exclusive: memory efficient or speed efficient. Memory-efficient algorithms vie to save as much memory as possible in order to have a Huffman codec running with very little memory. This is achieved by alternative encoding and decoding algorithms, as much as alternative data structures. Speed-efficient algorithms are concerned only with the speed of coding or decoding data with a Huffman codebook. This also relies on alternative data structures, but there is not necessarily a concern about how much memory is spent.

4.5.1 Memory-Efficient Algorithms

If we opt for an explicit tree representation to maintain the Huffman code information in memory, we have only a few choices. One is a classical binary tree, with three pointers (one for the parent, two for the children), plus possibly extra information. Another possible choice is to use heap-type storage (where the tree is stored in an array organized in a way that the parent for a node $0 \leq k$ is at $k/2$, and its children are at $2k + 1$ and $2k + 2$). This kind of storage is potentially sparse because while the Huffman code tree is full, it is rarely complete. Let the reader ponder the amount of memory potentially wasted by this method.

Moffat and Katajainen [16] proposed a dynamic programming algorithm to compute the lengths of the codes, given that the probabilities are sorted in non-increasing order, in $O(n)$, using $O(1)$ working space, as it overwrites the frequencies with the lengths as the algorithm progresses. This algorithm saves working space, especially all the housekeeping data that a list and tree-based algorithm would need. This leads to substantial savings when the number of symbols in the alphabet is large. In companion papers, Moffat and Turpin [29, 30] described an algorithm that allows us to use only $O(L_{\max})$ memory, using a few tables that describe not the entire tree, but only the different lengths of the codes and their number. Knowing how many codes there are of a given length is enough to compute the Huffman code of a symbol at encode time. Fortunately, it also works at decode time. The total memory needed by this algorithm is $O(L_{\max})$ machine words. This algorithm was also proposed by Hirschberg and Lelever [31], and it seems that it even predates this paper!

4.5.2 Speed-Efficient Algorithms

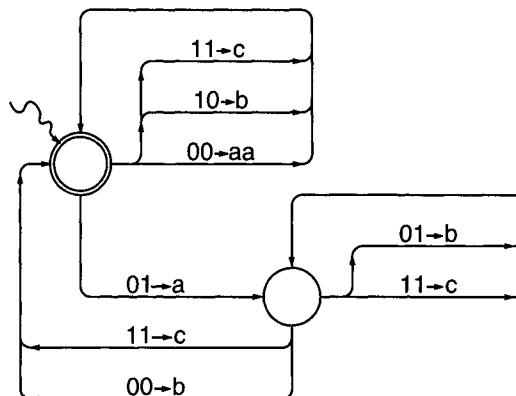
All algorithms presented so far proceed to encoding or decoding 1 bit at a time. To speed up decoding, Choueka *et al.* [32], Siemiński [33], and Tanaka [34] introduced various data structures and algorithms, all using the concept of finite-state automata. While the use of an automaton is not explicit in all papers, the basic idea is that rather than decoding bit by bit, decoding of the input will proceed by chunks of m bits, and the internal state of the decoder will be represented as the state of an augmented automaton.

An automaton, in the sense of the theory of computation, is composed of an input alphabet A , a set of states Σ , a set $F \subseteq \Sigma$ of accepting states, an initial state i , and a transition function $T : \Sigma \times A^* \rightarrow \Sigma$. The variety of automata considered here also has an output alphabet Ω , so the transition function becomes $T : \Sigma \times A^* \rightarrow \Sigma \times \Omega^*$. The transition function takes as input the current state and a certain number of input symbols, produces a certain number of output symbols, and changes the current state to a new state. The automaton starts in state i . The automaton is allowed to stop if the current state is one of the accepting states in F .

How this applies to fast Huffman decoding is not obvious. In this case, the input alphabet A is $\{0, 1\}$ and Ω is the original message alphabet. Since we are interested in reading m bits at a time, we will have to consider all possible strings of m bits and all their prefixes and suffixes. For example, let us consider the codebook $\{a \rightarrow 0, b \rightarrow 10, c \rightarrow 11\}$ and a decode block length of $m = 2$. This gives us four possible blocks, $\{00, 01, 10, 11\}$. Block 00 can be either aa or the end of b and a ; block 11 either is the end of the code of c and the beginning of b or c or is the code for c , depending on what was read before this block, what we call the context. The context is maintained by the automaton in the current state. Knowing the current state and the current input block is enough to unambiguously decode the input. The new state generated by the transition function will represent the new context, and decoding continues until we reach an accepting state and no more input symbols are to be read. Figure 4.5 shows the automaton for the simple code, and Table 4.3 shows its equivalent table form.

Table 4.3 The Table Representation of the Automaton in Fig. 4.5

State	Read	Emit	Goto
0	00	<i>aa</i>	0
	01	<i>a</i>	1
	10	<i>b</i>	0
	11	<i>c</i>	0
1	00	<i>ba</i>	0
	01	<i>b</i>	1
	10	<i>ca</i>	0
	11	<i>c</i>	1

**FIGURE 4.5**

A fast decode automaton for the code $\{a \rightarrow 0, b \rightarrow 10, c \rightarrow 11\}$. The squiggly arrow indicates the initial state. The accepting state (where the decoding is allowed to stop) is encircled by a double line. The labels on the automaton, such as $b_1b_2 \rightarrow s$, read as “on reading b_1b_2 , output string s ”.

As the reader may guess, the number of states and transitions grows rapidly with the number and length of codes, as well as the block size. Speed is gained at the expense of a great amount of memory. We can apply a minimizing algorithm to reduce the size of the automaton, but we still may have a great number of states and transitions. We also must consider the considerable computation needed to compute the automaton in a minimized form.

Other authors presented the idea of compressing a binary Huffman code tree to, say, a quaternary tree, effectively reducing by half the depth of the tree. This technique is due to Bassiouni and Mukherjee [35]. They also suggest using a fast decoding data structure, which is essentially an automaton by another name. Cheung *et al.* [36] proposed a fast decoding architecture based on a mixture of programmable logic, lookup tables, and software generation. The good thing about programmable logic is that it can be programmed on the fly, so the encoder/decoder can be made optimal with respect to the source. Current programmable logic technologies are not as fast as current VLSI technologies, but what they lack in speed they make up for in flexibility.

However, none of these few techniques takes into account the case where the Huffman codes are dynamic. Generating an automaton each time the code adapts is of course computationally prohibitive, even for small message alphabets. Reconfiguring programmable logic still takes a long time (it takes on the order of seconds), so one cannot consider reprogramming very often.

4.6 CONCLUSION AND FURTHER READING

The literature on Huffman coding is so abundant that an exhaustive list of papers, books, and articles is all but impossible to compile. We note a large number of papers concerned with efficient implementations, in the contexts of data communications, compressed file systems, archival, and search. For that very reason, there are a great number of topics related to Huffman coding that we could not present here, but we can encourage the reader to look for more papers and books on this rich topic.

Among the topics we have not discussed is *alphabetical coding*. An alphabetical code is a code such that if a precedes b , noted $a \prec b$, in the original alphabet, then $\text{code}(a)$ lexicographically precedes $\text{code}(b)$ when the codes are not considered as numerical values but as strings of output symbols. Note that here, lexicographically ordered does not have the same meaning as that given in Section 4.4.2, where we discussed algorithm FGK. This property allows, for example, direct comparison of compressed strings. That is, if $aa \prec ab$, then $\text{code}(aa) \prec \text{code}(ab)$. Alphabetical codes are therefore suitable for compressing individual strings in an index or other search table. The reader may want to consult the papers by Gilbert and Moore [37], by Yeung [38], and by Hu and Tucker [39], where algorithms to build minimum-redundancy alphabetical codes are presented.

Throughout this chapter, we assumed that all output symbols had an equal cost. Varn presents an algorithm where output symbols are allowed to have a cost [40]. This situation arises when, say, the communication device spends more power on some symbols than others. Whether the device uses many different, but fixed, voltage levels or encodes the signal differentially, as in the LVDS standard, there are symbols that cost more in power to emit than others. Obviously, we want to minimize both communication time and power consumption. Varn's algorithm lets us optimize a code considering the relative cost of the output symbols. This is critical for low-powered devices such as handheld PDAs, cellular phones, and the like.

In conclusion, let us point out that Huffman codes are rarely used as the sole means of compression. Many more elaborate compression algorithms use Huffman codes to compress pointers, integers, and other side information that they need to store efficiently to achieve further compression. We only have to think of compression algorithms such as the celebrated sliding window Ziv–Lempel technique that encodes both the position and the length of a match as its mean of compression. Using naive representations for these quantities leads to diminished compression, while using Huffman codes give much more satisfactory results.

4.7 REFERENCES

1. Golomb, S. W., 1966. Run length encodings. *IEEE Transactions on Information Theory*, Vol. 12, No. 3, pp. 399–401.
2. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, pp. 490–505, April 1989.
3. Huffman, D., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the I.R.E.*, Vol. 40, pp. 1098–1101.
4. Fano, R. M., 1944. The Transmission of Information, Technical Report 65, Research Laboratory of Electronics, MIT, Cambridge, MA.
5. Gallager, R. G., 1978. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, Vol. 24, pp. 668–674, November 1978.
6. Capocelli, R. M., R. Giancarlo, and I. J. Taneja, 1996. Bounds on the redundancy of Huffman codes. *IEEE Transactions on Information Theory*, Vol. 32, No. 6, pp. 854–857.
7. Buro, M., 1993. On the maximum length of Huffman codes. *Information Processing Letters*, Vol. 45, pp. 219–223.

8. Campos, A. S. E., 2000. When Fibonacci and Huffman Met. Available at http://www.arturocampos.com/ac_fib_Huffman.html.
9. Katona, G. O. H., and T. O. H. Nemetz, 1976. Huffman codes and self-information. *IEEE Transactions on Information Theory*, Vol. 22, No. 3, pp. 337–340.
10. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*. Prentice-Hall, New York. [QA76.9 T48B45].
11. Tunstall, B. P., 1967. *Synthesis of Noiseless Compression Codes*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, September 1967.
12. Welch, T. A., 1894. A technique for high performance data compression. *Computer*, pp. 8–19, June 1894.
13. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, pp. 530–536, September 1978.
14. Fraenkel, A. S., and S. T. Klein, 1993. Bounding the depth of search trees. *The Computer Journal*, Vol. 36, No. 7, pp. 668–678.
15. Milidiú, R. L., A. A. Pessoa, and E. S. Laber, 1998. In-place length-restricted prefix coding. In *String Processing and Information Retrieval*, pp. 50–59.
16. Moffat, A., and J. Katajainen, 1995. In place calculation of minimum redundancy codes. In *Proceedings of the Workshop on Algorithms and Data Structures*, pp. 303–402, Springer-Verlag, Berlin/New York. [LNCS 955].
17. Larimore, L. L., and D. S. Hirschberg, 1990. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, Vol. 37, pp. 464–473, July 1990.
18. Faller, N., 1973. An adaptive system for data compression. In *Records of the 7th Asilomar Conference on Circuits, Systems and Computers*, pp. 393–397.
19. Cormack, G. V., and R. N. Horspool, 1984. Algorithms for adaptive Huffman codes. *Information Processing Letters*, Vol. 18, pp. 159–165.
20. Knuth, D. E., 1983. Dynamic Huffman coding. *Journal of Algorithms*, Vol. 6, pp. 163–180.
21. Vitter, J. S., 1987. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, Vol. 34, pp. 823–843, October 1987.
22. Vitter, J. S., 1989. Algorithm 673: Dynamic Huffman coding. *ACM Transactions on Mathematics Software*, Vol. 15, pp. 158–167, June 1989.
23. Jones, D. W., 1988. Application of splay trees to data compression. *Communications of the ACM*, Vol. 31, pp. 996–1007.
24. Tarjan, R. E., 1983. *Data Structures and Network Algorithms*, No. 44, Regional Conference Series in Applied Mathematics, SIAM.
25. Pigeon, S., and Y. Bengio, 1997. A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Technical Report 1081, Département d’Informatique et Recherche Opérationnelle, Université de Montréal, Montréal, Québec, Canada.
26. Pigeon, S., and Y. Bengio, 1997. A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, Revisited, Technical Report 1095, Département d’Informatique et Recherche Opérationnelle, Université de Montréal, Montréal, Québec, Canada.
27. Pigeon, S., and Y. Bengio, 1998. Memory-efficient adaptive Huffman coding. *Doctor Dobb’s Journal*, No. 290, pp. 131–135.
28. McIntyre, D. R., and M. A. Pechura, 1985. Data compression using static Huffman code–decode tables. *Communications of the ACM*, Vol. 28, pp. 612–616, June 1985.
29. Moffat, A., and A. Turpin, 1997. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, Vol. 45, pp. 1200–1207, October 1997.
30. Moffat, A., and A. Turpin, 1996. On the implementation of minimum redundancy prefix codes [extended abstract]. In *Proceedings of the Data Compression Conference* (M. C. James A. Storer, Ed.), pp. 170–179, IEEE Comput. Soc., Los Alamitos, CA.
31. Hirschberg, D. S., and D. A. Lelewer, 1990. Efficient decoding of prefix codes. *Communications of the ACM*, Vol. 33, No. 4, pp. 449–459.
32. Choueka, Y., S. T. Klein, and Y. Perl, 1985. Efficient variants of Huffman codes in high level languages. In *Proceedings of the 8th ACM-SIGIR Conference, Montreal*, pp. 122–130.

33. Siemiński, A., 1988. Fast decoding of the Huffman codes. *Information Processing Letters*, Vol. 26, pp. 237–241, January 1988.
34. Tanaka, H., 1987. Data structure of Huffman codes and its application to efficient coding and decoding. *IEEE Transactions on Information Theory*, Vol. 33, pp. 154–156, January 1987.
35. Bassiouni, M. A., and A. Mukherjee, 1995. Efficient decoding of compressed data. *Journal of the American Society of Information Science*, Vol. 46, No. 1, pp. 1–8.
36. Cheung, G., S. McCanne, and C. Papadimitriou, 1999. Software synthesis of variable-length code decoder using a mixture of programmed logic and table lookups. In *Proceedings of the Data Compression Conference* (M. C. James A. Storer, Ed.), pp. 121–139, IEEE Comput. Soc., Los Alamitos, CA.
37. Gilbert, E. N., and E. F. Moore, 1959. Variable length binary encodings. *Bell Systems Technical Journal*, Vol. 38, pp. 933–968.
38. Yeung, R. W., 1991. Alphabetic codes revisited. *IEEE Transactions on Information Theory*, Vol. 37, pp. 564–572, May 1991.
39. Hu, T. C., and A. C. Tucker, 1972. Optimum computer search trees and variable length alphabetic codes. *SIAM*, Vol. 22, pp. 225–234.
40. Varn, B., 1971. Optimal variable length codes (arbitrary symbol cost and equal word probability). *Information and Control*, Vol. 19, pp. 289–301.

This Page Intentionally Left Blank

Arithmetic Coding

AMIR SAID

OVERVIEW

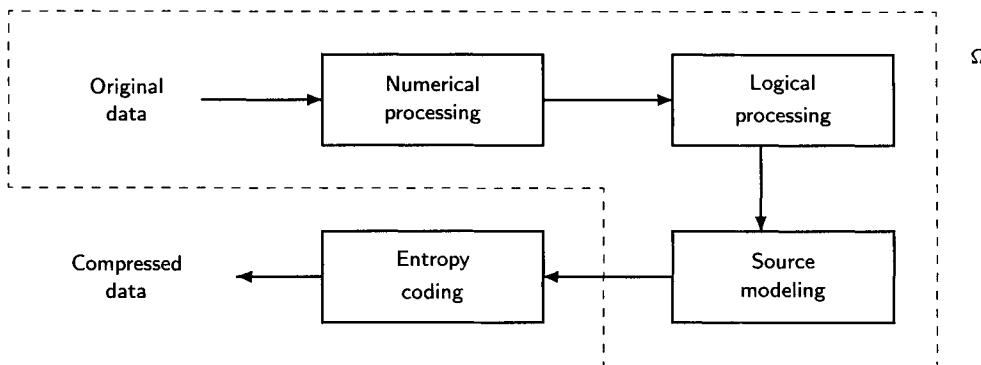
This introduction to arithmetic coding is divided in two parts. The first explains how and why arithmetic coding works. We start presenting it in very general terms, so that its simplicity is not lost under layers of implementation details. Next, we show some of its basic properties, which are later used in the computational techniques required for a practical implementation.

In the second part, we cover the practical implementation aspects, including arithmetic operations with low precision, the subdivision of coding and modeling, and the realization of adaptive encoders. We also analyze the arithmetic coding computational complexity and techniques to reduce it.

We start some sections by first introducing the notation and most of the mathematical definitions. The reader should not be intimidated if at first their motivation is not clear: These are always followed by examples and explanations.

5.1 INTRODUCTION

Compression applications employ a wide variety of techniques and have quite different degrees of complexity, but share some common processes. Figure 5.1 shows a diagram with typical processes used for data compression. These processes depend on the data type, and the blocks in Fig. 5.1 may be in a different order or combined. Numerical processing, like predictive coding and linear transforms, is normally used for waveform signals, like images and audio [20, 35, 36, 48, 55]. Logical processing consists of changing the data to a form more suitable for compression, like run-lengths, zero-trees, set-partitioning information, and dictionary entries [3, 20, 38, 40, 41, 43, 47, 55]. The next stage, source modeling, is used to account for variations in the statistical

**FIGURE 5.1**

System with typical processes for data compression. Arithmetic coding is normally the final stage, and the other stages can be modeled as a single data source Ω .

properties of the data. It is responsible for gathering statistics and identifying data contexts that make the source models more accurate and reliable [14, 28, 29, 45, 46, 49, 53].

What most compression systems have in common is the fact that the final process is *entropy coding*, i.e., the process of representing information in the most compact form. It may be responsible for doing most of the compression work or it may just complement what has been accomplished by previous stages.

When we consider all the different entropy-coding methods, and their possible applications in compression applications, one method is able to work most efficiently in the largest number of circumstances and purposes and stands out in terms of elegance, effectiveness, and versatility: arithmetic coding. Among its most desirable features we have the following.

- When applied to independent and identically distributed (i.i.d.) sources, the compression of each symbol is provably optimal.
- It simplifies automatic modeling of complex sources, yielding near-optimal or significantly improved compression for sources that are not i.i.d.
- It is effective in a wide range of situations and compression ratios. The same arithmetic coding implementation can effectively code all the diverse data created by the different processes of Fig. 5.1, such as modeling parameters, transform coefficients, and signaling.
- Its main process is arithmetic, which is supported with ever-increasing efficiency by all general-purpose or digital signal processors (CPUs, DSPs) [58–61].
- It is suitable for use as a “compression black-box” by those that are not coding experts or do not want to implement the coding algorithm themselves.

Even with all these advantages, arithmetic coding is not as popular and well understood as other methods. Certain practical problems held back its adoption.

- The complexity of arithmetic operations was excessive for coding applications.
- Patents covered the most efficient implementations. Royalties and the fear of patent infringement discouraged arithmetic coding in commercial products.
- Efficient implementations were difficult to understand.

However, these issues are now mostly overcome. First, the relative efficiency of computer arithmetic improved dramatically, and new techniques avoid the most expensive operations.

Second, many of the patents have expired (e.g., [11, 16]), or became obsolete. Finally, we do not need to worry so much about complexity-reduction details that obscure the inherent simplicity of the method. Current computational resources allow us to implement simple, efficient, and royalty-free arithmetic coding.

5.2 BASIC PRINCIPLES

5.2.1 Notation

Let Ω be a data source that puts out symbols s_k coded as integer numbers in the set $\{0, 1, \dots, M - 1\}$, and let $S = \{s_1, s_2, \dots, s_N\}$ be a sequence of N random symbols put out by Ω [1, 4, 5, 21, 55, 56]. For now, we assume that the source symbols are independent and identically distributed [22], with probability

$$p(m) = \text{Prob}\{s_k = m\}, \quad m = 0, 1, 2, \dots, M - 1, \quad k = 1, 2, \dots, N. \quad (5.1)$$

We also assume that for all symbols we have $p(m) \neq 0$, and define $c(m)$ to be the cumulative distribution

$$c(m) = \sum_{s=0}^{m-1} p(s), \quad m = 0, 1, \dots, M. \quad (5.2)$$

Note that $c(0) \equiv 0$, $c(M) \equiv 1$, and

$$p(m) = c(m + 1) - c(m). \quad (5.3)$$

We use boldface letters to represent the vectors with all $p(m)$ and $c(m)$ values, i.e.,

$$\begin{aligned} \mathbf{p} &= [p(0) \ p(1) \ \cdots \ p(M - 1)], \\ \mathbf{c} &= [c(0) \ c(1) \ \cdots \ c(M)]. \end{aligned}$$

We assume that the compressed data (output of the encoder) is saved in a vector (buffer) \mathbf{d} . The output alphabet has D symbols i.e., each element in \mathbf{d} is number in the set $\{0, 1, \dots, D - 1\}$.

Under the assumptions above, an optimal coding method [1] codes each symbol s from Ω with an average number of bits equal to

$$B(s) = -\log_2 p(s) \quad \text{bits.} \quad (5.4)$$

Example 5.1. Data source Ω can be a file with English text: each symbol from this source is a single byte with the ASCII (American Standards Committee for Information Interchange) representation of a character. This data alphabet contains $M = 256$ symbols, and symbol numbers are defined by the ASCII standard. The probabilities of the symbols can be estimated by gathering statistics using a large number of English texts. Table 5.1 shows some characters, their ASCII symbol values, and their estimated probabilities. It also shows the number of bits required to code symbol s in an optimal manner, $-\log_2 p(s)$. From these numbers we conclude that, if data symbols in English text were i.i.d., then the best possible text compression ratio would be about 2:1 (4 bits/symbol). Specialized text compression methods [8, 10, 29, 41] can yield significantly better compression ratios because they exploit the statistical dependence between letters.

This first example shows that our initial assumptions about data sources are rarely found in practical cases. More commonly, we have the following issues.

Table 5.1 Estimated Probabilities of Some Letters and Punctuation Marks in the English Language

Character	ASCII Symbol <i>s</i>	Probability <i>p(s)</i>	Optimal Number of Bits $-\log_2 p(s)$
Space	32	0.1524	2.714
,	44	0.0136	6.205
.	46	0.0056	7.492
A	65	0.0017	9.223
B	66	0.0009	10.065
C	67	0.0013	9.548
a	97	0.0595	4.071
b	98	0.0119	6.391
c	99	0.0230	5.441
d	100	0.0338	4.887
e	101	0.1033	3.275
f	102	0.0227	5.463
t	116	0.0707	3.823
z	122	0.0005	11.069

Note: Symbols are numbered according to the ASCII standard.

1. The source symbols are not identically distributed.
2. The symbols in the data sequence are not independent (even if uncorrelated) [22].
3. We can only *estimate* the probability values, the statistical dependence between symbols, and how they change in time.

However, in the next sections we show that the generalization of arithmetic coding to time-varying sources is straightforward, and we explain how to address all these practical issues.

5.2.2 Code Values

Arithmetic coding is different from other coding methods for which we know the exact relationship between the coded symbols and the actual bits that are written to a file. It codes one data symbol at a time and assigns to each symbol a real-valued number of bits (see examples in Table 5.1). To figure out how this is possible, we must understand the *code value* representation: coded messages mapped to real numbers in the interval $[0, 1]$.

The code value v of a compressed data sequence is the real number with fractional digits equal to the sequence's symbols. We can convert sequences to code values by simply adding “0.” to the beginning of a coded sequence and then interpreting the result as a number in base- D notation, where D is the number of symbols in the coded sequence alphabet. For example, if a coding method generates the sequence of bits 0011000101100, then we have

$$\begin{aligned} \text{Code sequence } \mathbf{d} &= [0011000101100] \\ \text{Code value } v &= 0.\overbrace{0011000101100}_2 = 0.19287109375, \end{aligned} \tag{5.5}$$

where the “2” subscript denotes base-2 notation. As usual, we omit the subscript for decimal notation.

This construction creates a convenient mapping between infinite sequences of symbols from a D -symbol alphabet and real numbers in the interval $[0, 1)$, where any data sequence can be represented by a real number and vice versa. The code value representation can be used for any coding system and it provides a universal way to represent large amounts of information independently of the set of symbols used for coding (binary, ternary, decimal, etc.). For instance, in (5.5) we see the same code with base-2 and base-10 representations.

We can evaluate the efficacy of any compression method by analyzing the distribution of the code values it produces. From Shannon's information theory [1] we know that if a coding method is optimal, then the cumulative distribution [22] of its code values must be a straight line from point $(0, 0)$ to point $(1, 1)$.

Example 5.2. Let us assume that the i.i.d. source Ω has four symbols and the probabilities of the data symbols are $\mathbf{p} = [0.65 \ 0.2 \ 0.1 \ 0.05]$. If we code random data sequences from this source with 2 bits per symbols, the resulting code values produce a cumulative distribution as shown in Fig. 5.2, under the label "uncompressed." Note how the distribution is skewed, indicating the possibility for significant compression.

The same sequences can be coded with the Huffman code for Ω [2, 4, 21, 55, 56], with 1 bit used for symbol "0", 2 bits for symbol "1", and 3 bits for symbols "2" and "3". The corresponding code value cumulative distribution in Fig. 5.2 shows that there is substantial improvement over the

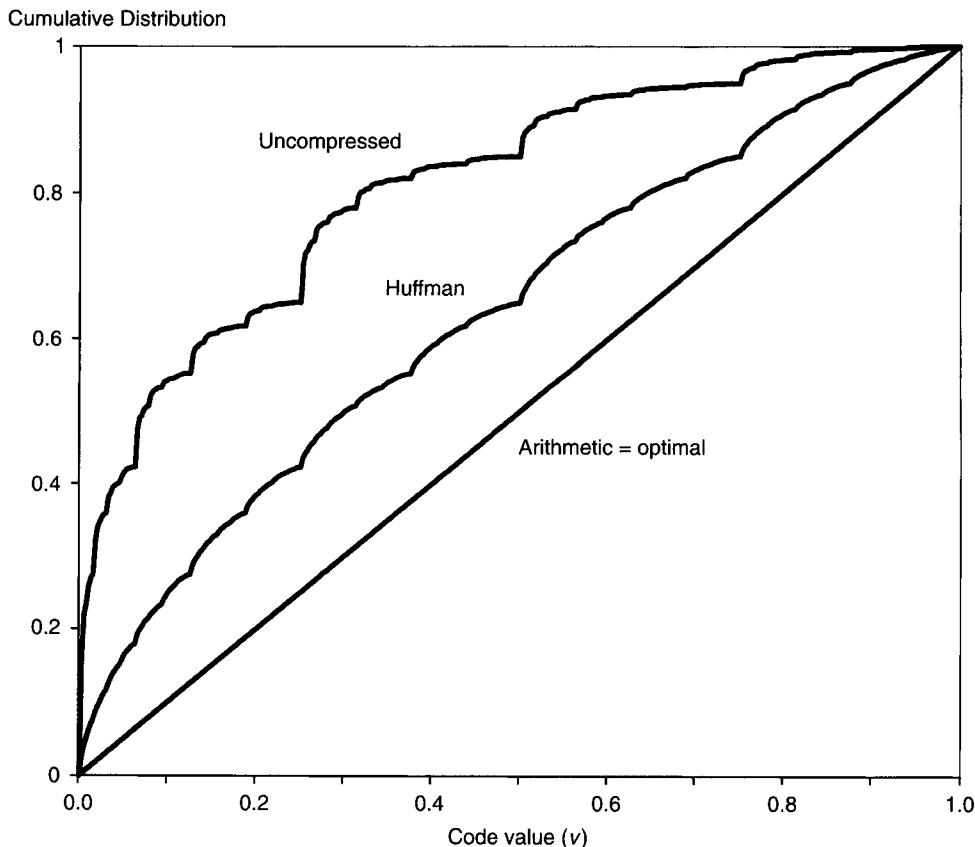


FIGURE 5.2

Cumulative distribution of code values generated by different coding methods when applied to the source of Example 5.2.

uncompressed case, but this coding method is still clearly not optimal. The third line in Fig. 5.2 shows that the sequences compressed with arithmetic coding simulation produce a code value distribution that is practically identical to the optimal.

The straight-line distribution means that if a coding method is optimal, then there is no statistical dependence or redundancy left in the compressed sequences, and consequently its code values are uniformly distributed on the interval [0, 1). This fact is essential for understanding of how arithmetic coding works. Moreover, code values are an integral part of the arithmetic encoding/decoding procedures, with arithmetic operations applied to real numbers that are directly related to code values.

One final comment about code values: Two infinitely long different sequences can correspond to the same code value. This follows from the fact that for any $D > 1$ we have

$$\sum_{n=k}^{\infty} (D-1)D^{-n} = D^{1-k}. \quad (5.6)$$

For example, if $D = 10$ and $k = 2$, then (5.6) is the equality $0.09999999\dots = 0.1$. This fact has no important practical significance for coding purposes, but we need to take it into account when studying some theoretical properties of arithmetic coding.

5.2.3 Arithmetic Coding

5.2.3.1 Encoding Process

In this section we first introduce the notation and equations that describe arithmetic encoding, followed by a detailed example. Fundamentally, the arithmetic encoding process consists of creating a sequence of nested intervals in the form $\Phi_k(S) = [\alpha_k, \beta_k]$, $k = 0, 1, \dots, N$, where S is the source data sequence, α_k and β_k are real numbers such that $0 \leq \alpha_k \leq \alpha_{k+1}$, and $\beta_{k+1} \leq \beta_k \leq 1$. For a simpler way to describe arithmetic coding we represent intervals in the form $|b, l\rangle$, where b is called the *base* or *starting point* of the interval, and l is the *length* of the interval. The relationship between the traditional and the new interval notation is

$$|b, l\rangle = [\alpha, \beta] \quad \text{if } b = \alpha \quad \text{and} \quad l = \beta - \alpha. \quad (5.7)$$

The intervals used during the arithmetic coding process are, in this new notation, defined by the set of recursive equations [5, 13]

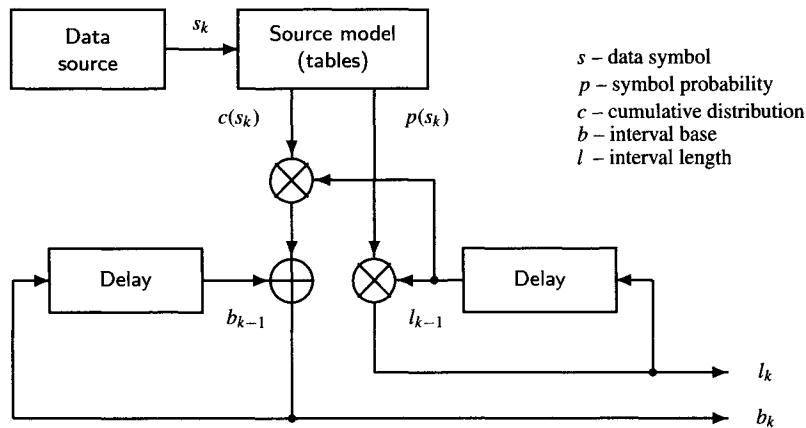
$$\Phi_0(S) = |b_0, l_0\rangle = |0, 1\rangle, \quad (5.8)$$

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c(s_k)l_{k-1}, p(s_k)l_{k-1}\rangle, \quad k = 1, 2, \dots, N. \quad (5.9)$$

The properties of the intervals guarantee that $0 \leq b_k \leq b_{k+1} < 1$, and $0 < l_{k+1} < l_k \leq 1$. Figure 5.3 shows a dynamic system corresponding to the set of recursive equations (5.9). We later explain how to choose, at the end of the coding process, a code value in the final interval, i.e., $\hat{v}(S) \in \Phi_N(S)$.

The coding process defined by (5.8) and (5.9), also called *Elias coding*, was first described in [5] (for another description of Elias coding see Chapter 1). Our convention of representing an interval using its base and length has been used since the first arithmetic coding papers [12, 13]. Other authors have intervals represented by their extreme points, like [base, base+length], but there is no mathematical difference between the two notations.

Example 5.3. Let us assume that source Ω has four symbols ($M = 4$), the probabilities and distribution of the symbols are $\mathbf{p} = [0.2 \ 0.5 \ 0.2 \ 0.1]$ and $\mathbf{c} = [0 \ 0.2 \ 0.7 \ 0.9 \ 1]$, and the sequence of ($N = 6$) symbols to be encoded is $S = \{2, 1, 0, 0, 1, 3\}$.

**FIGURE 5.3**

Dynamic system for updating arithmetic coding intervals.

Figure 5.4 shows graphically how the encoding process corresponds to the selection of intervals in the line of real numbers. We start at the top of the figure, with the interval $[0, 1]$, which is divided into four subintervals, each with length equal to the probability of the data symbols. Specifically, interval $[0, 0.2)$ corresponds to $s_1 = 0$, interval $[0.2, 0.7)$ corresponds to $s_1 = 1$, interval $[0.7, 0.9)$ corresponds to $s_1 = 2$, and finally interval $[0.9, 1)$ corresponds to $s_1 = 3$. The next set of allowed nested subintervals also have length proportional to the probability of the symbols, but their lengths are also proportional to the length of the interval they belong to. Furthermore, they represent more than one symbol value. For example, interval $[0, 0.04)$ corresponds to $s_1 = 0, s_2 = 0$, interval $[0.04, 0.14)$ corresponds to $s_1 = 0, s_2 = 1$, and so on.

The interval lengths are reduced by factors equal to symbol probabilities in order to obtain code values that are uniformly distributed in the interval $[0, 1]$ (a necessary condition for optimality, as explained in Section 5.2.2). For example, if 20% of the sequences start with symbol “0”, then 20% of the code values must be in the interval assigned to those sequences, which can be achieved only if we assign to the first symbol “0” an interval with length equal to its probability, 0.2. The same reasoning applies to the assignment of the subinterval lengths: Every occurrence of symbol “0” must result in a reduction of the interval length to 20% of its current length. This way, after several symbols are encoded, the distribution of code values should be a very good approximation of a uniform distribution.

Equations (5.8) and (5.9) provide the formulas for the sequential computation of the intervals. Applying them to our example we obtain:

$$\Phi_0(S) = [0, 1] = [0, 1],$$

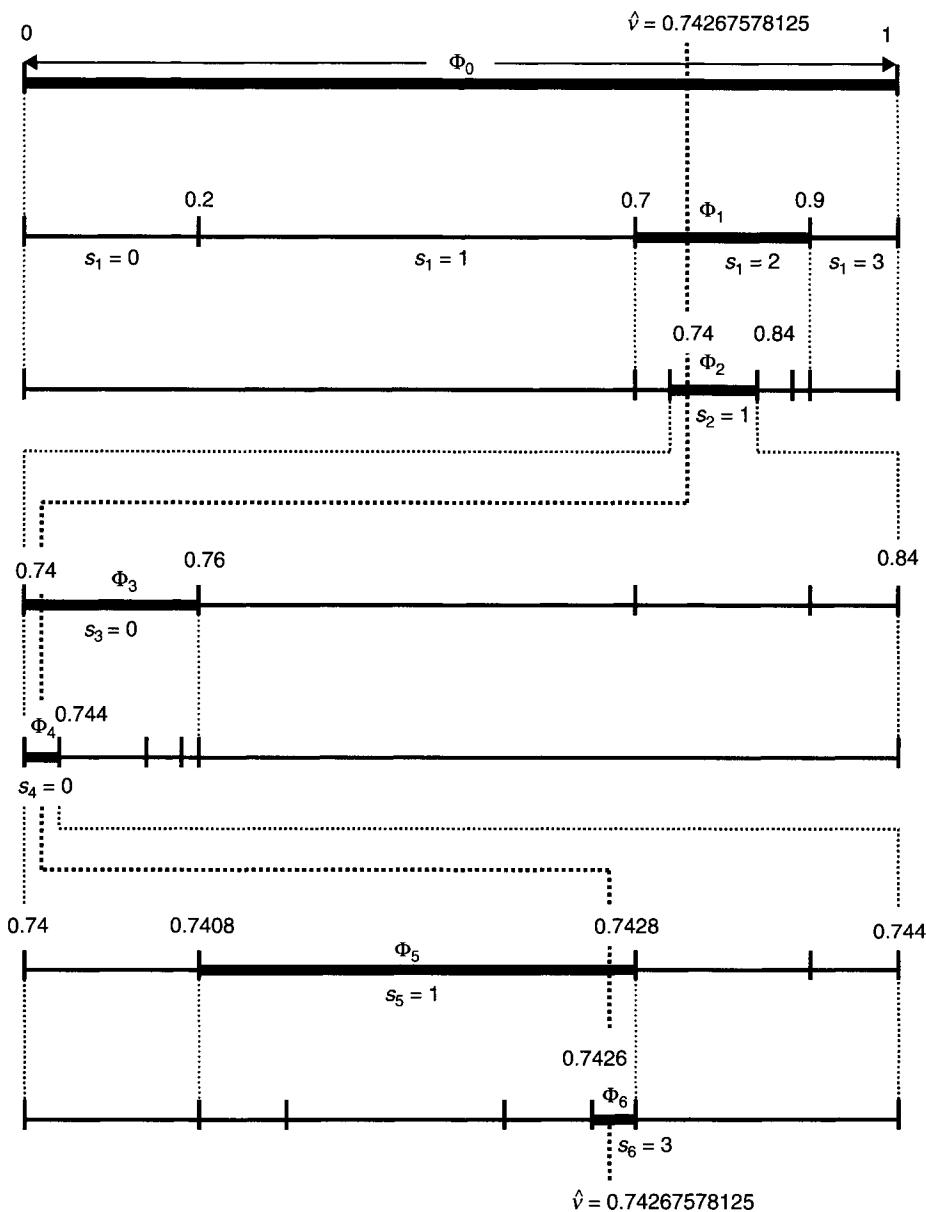
$$\Phi_1(S) = [b_0 + c(2)l_0, p(2)l_0] = [0 + 0.7 \times 1, 0.2 \times 1] = [0.7, 0.9],$$

$$\Phi_2(S) = [b_1 + c(1)l_1, p(1)l_1] = [0.7 + 0.2 \times 0.2, 0.5 \times 0.2] = [0.74, 0.84],$$

 \vdots

$$\Phi_6(S) = [b_5 + c(3)l_5, p(3)l_5] = [0.7426, 0.0002] = [0.7426, 0.7428].$$

The list with all the encoder intervals is shown in Table 5.2. Since the intervals quickly become quite small, in Fig. 5.4 we must graphically magnify them (twice) so that we can see how the coding

**FIGURE 5.4**

Graphical representation of the arithmetic coding process of Example 5.3: The interval $\Phi_0 = [0, 1]$ is divided into nested intervals according to the probability of the data symbols. The selected intervals, corresponding to data sequence $S = \{2, 1, 0, 0, 1, 3\}$, are indicated by thicker lines.

process continues. Note that even though the intervals are shown in different magnifications, the interval values do not change, and the interval subdivision process continues in exactly the same manner.

The final task in arithmetic encoding is to define a code value $\hat{v}(S)$ that will represent data sequence S . In the next section we show how the decoding process works correctly for *any* code

Table 5.2 Arithmetic Encoding and Decoding Results for Examples 5.3 and 5.4

Iteration k	Input Symbol s_k	Interval Base b_k	Interval Length l_k	Decoder Updated Value $\hat{v}_k = \frac{\hat{v} - b_{k-1}}{l_{k-1}}$	Output Symbol \hat{s}_k
0	—	0	1	—	—
1	2	0.7	0.2	0.74267578125	2
2	1	0.74	0.1	0.21337890625	1
3	0	0.74	0.02	0.0267578125	0
4	0	0.74	0.004	0.1337890625	0
5	1	0.7408	0.002	0.6689453125	1
6	3	0.7426	0.0002	0.937890625	3
7	—	—	—	0.37890625	1
8	—	—	—	0.3578125	1

Note: The last two rows show what happens when decoding continues past the last symbol.

value $\hat{v} \in \Phi_N(S)$. However, the code value cannot be provided to the decoder as a pure real number. It must be stored or transmitted, using a conventional number representation. Since we have the freedom to choose any value in the final interval, we want to choose the values with the shortest representation. For instance, in Example 5.3, the shortest decimal representation comes from choosing $\hat{v} = 0.7427$, and the shortest binary representation is obtained with $\hat{v} = 0.1011110001_2 = 0.74267578125$.

The process to find the best binary representation is quite simple and best shown by induction. The main idea is that for relatively large intervals we can find the optimal value by testing a few binary sequences, and as the interval lengths are halved, the number of sequences to be tested must double, increasing the number of bits by 1. Thus, according to the interval length l_N , we use the following rules:

- If $l_N \in [0.5, 1)$, then choose code value $\hat{v} \in \{0, 0.5\} = \{0.0_2, 0.1_2\}$ for a 1-bit representation.
- If $l_N \in [0.25, 0.5)$, then choose value $\hat{v} \in \{0, 0.25, 0.5, 0.75\} = \{0.00_2, 0.01_2, 0.10_2, 0.11_2\}$ for a 2-bit representation.
- If $l_N \in [0.125, 0.25)$, then choose value $\hat{v} \in \{0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875\} = \{0.000_2, 0.001_2, 0.010_2, 0.011_2, 0.100_2, 0.101_2, 0.110_2, 0.111_2\}$ for a 3-bit representation.

By observing the pattern we conclude that the minimum number of bits required for representing $\hat{v} \in \Phi_N(S)$ is

$$B_{\min} = \lceil -\log_2(l_N) \rceil \text{ bits,} \quad (5.10)$$

where $\lceil x \rceil$ represents the smallest integer greater than or equal to x .

We can test this conclusion observing the results for Example 5.3 in Table 5.2. The final interval is $l_N = 0.0002$, and thus $B_{\min} = \lceil -\log_2(0.0002) \rceil = 13$ bits. However, in Example 5.3 we can choose $\hat{v} = 0.1011110001_2$, and it requires only 11 bits!

The origin of this inconsistency is the fact that we can choose binary representations with the number of bits given by (5.10) and then remove the trailing zeros. However, with optimal coding the *average* number of bits that can be saved with this process is only 1 bit, and for that reason, it is rarely applied in practice.

5.2.3.2 Decoding Process

In arithmetic coding, the decoded sequence is determined solely by the code value \hat{v} of the compressed sequence. For that reason, we represent the decoded sequence as

$$\hat{S}(\hat{v}) = \{\hat{s}_1(\hat{v}), \hat{s}_2(\hat{v}), \dots, \hat{s}_N(\hat{v})\}. \quad (5.11)$$

We now show the decoding process by which any code value $\hat{v} \in \Phi_N(S)$ can be used for decoding the correct sequence (i.e., $\hat{S}(\hat{v}) = S$). We present the set of recursive equations that implement decoding, followed by a practical example that provides an intuitive idea of how the decoding process works and why it is correct.

The decoding process recovers the data symbols in the same sequence that they were coded. Formally, to find the numerical solution, we define a sequence of normalized code values $\{\tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_N\}$. Starting with $\tilde{v}_1 = \hat{v}$, we sequentially find \hat{s}_k from \tilde{v}_k and then we compute \tilde{v}_{k+1} from \hat{s}_k and \tilde{v}_k .

The recursion formulas are

$$\tilde{v}_1 = \hat{v}, \quad (5.12)$$

$$\hat{s}_k(\hat{v}) = \{s : c(s) \leq \tilde{v}_k < c(s + 1)\}, \quad k = 1, 2, \dots, N, \quad (5.13)$$

$$\tilde{v}_{k+1} = \frac{\tilde{v}_k - c(\hat{s}_k(\hat{v}))}{p(\hat{s}_k(\hat{v}))}, \quad k = 1, 2, \dots, N - 1. \quad (5.14)$$

(In Eq. (5.13) the colon means “ s that satisfies the inequalities.”)

A mathematically equivalent decoding method—which later we show to be necessary when working with fixed-precision arithmetic—recovers the sequence of intervals created by the encoder and searches for the correct value $\hat{s}_k(\hat{v})$ in each of these intervals. It is defined by

$$\Phi_0(S) = |b_0, l_0\rangle = |0, 1\rangle, \quad (5.15)$$

$$\hat{s}_k(\hat{v}) = \left\{ s : c(s) \leq \frac{\hat{v} - b_{k-1}}{l_{k-1}} < c(s + 1) \right\}, \quad k = 1, 2, \dots, N, \quad (5.16)$$

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c(\hat{s}_k(\hat{v}))l_{k-1}, p(\hat{s}_k(\hat{v}))l_{k-1}\rangle, \quad k = 1, 2, \dots, N. \quad (5.17)$$

The combination of recursion (5.14) with recursion (5.17) yields

$$\tilde{v}_k = \frac{\hat{v} - \sum_{i=1}^{k-1} c(\hat{s}_i) \prod_{j=1}^{i-1} p(\hat{s}_j)}{\prod_{i=1}^{k-1} p(\hat{s}_i)} = \frac{\hat{v} - b_{k-1}}{l_{k-1}}. \quad (5.18)$$

showing that (5.13) is equivalent to (5.16).

Example 5.4. Let us apply the decoding process to the data obtained in Example 5.3. In Fig. 5.4, we show graphically the meaning of \hat{v} : It is a value that belongs to all nested intervals created during coding. The dotted line shows that its position moves as we magnify the graphs, but the value remains the same. From Fig. 5.4, we can see that we can start decoding from the first interval $\Phi_0(S) = [0, 1]$: We just have to compare \hat{v} with the cumulative distribution c to find the only possible value of \hat{s}_1

$$\hat{s}_1(\hat{v}) = \{s : c(s) \leq \hat{v} = 0.74267578125 < c(s + 1)\} = 2.$$

We can use the value of \hat{s}_1 to find out interval $\Phi_1(S)$ and use it for determining \hat{s}_2 . In fact, we can “remove” the effect of \hat{s}_1 in \hat{v} by defining the normalized code value

$$\tilde{v}_2 = \frac{\hat{v} - c(\hat{s}_1)}{p(\hat{s}_1)} = 0.21337890625.$$

Note that, in general, $\tilde{v}_2 \in [0, 1)$; i.e., it is a value normalized to the initial interval. In this interval we can use the same process to find

$$\hat{s}_2(\hat{v}) = \{s : c(s) \leq \tilde{v}_2 = 0.21337890625 < c(s + 1)\} = 1.$$

The last columns of Table 5.2 show how the process continues and the updated values are computed while decoding. We could say that the process continues until \hat{s}_6 is decoded. However, how can the decoder, having only the initial code value \hat{v} , know that it is time to stop decoding? The answer is simple: It cannot. We added two extra rows to Table 5.2 to show that the decoding process can continue normally after the last symbol is encoded. Below we explain what happens.

It is important to understand that arithmetic encoding maps intervals to *sets of sequences*. Each real number in an interval corresponds to one infinite sequence. Thus, the sequences corresponding to $\Phi_6(S) = [0.7426, 0.7428)$ are all those that *start* as $\{2, 1, 0, 0, 1, 3, \dots\}$. The code value $\hat{v} = 0.74267578125$ corresponds to one such infinite sequence, and the decoding process can go on forever decoding that particular sequence.

There are two practical ways to indicate that decoding should stop:

1. Provide the number of data symbols (N) in the beginning of the compressed file.
2. Use a special symbol as “end-of-message,” which is coded only at the end of the data sequence, and assign to this symbol the smallest probability value allowed by the encoder/decoder.

As we explained above, the decoding procedure will always produce a decoded data sequence. However, how do we know that it is the right sequence? This can be inferred from the fact that if S and S' are sequences with N symbols, then

$$S \neq S' \Leftrightarrow \Phi_N(S) \cap \Phi_N(S') = \emptyset. \quad (5.19)$$

This guarantees that different sequences cannot produce the same code value. In Section 5.2.5.6 we show that, due to approximations, we have incorrect decoding if (5.19) is not satisfied.

5.2.4 Optimality of Arithmetic Coding

Information theory [1, 4, 5, 21, 32, 55, 56] shows us that the average number of bits needed to code each symbol from a stationary and memoryless source Ω cannot be smaller than its entropy $H(\Omega)$, defined by

$$H(\Omega) = - \sum_{m=0}^{M-1} p(m) \log_2 p(m) \quad \text{bits/symbol.} \quad (5.20)$$

We have seen that the arithmetic coding process generates code values that are uniformly distributed across the interval $[0, 1)$. This is a necessary condition for optimality, but not a sufficient one. In the interval $\Phi_N(S)$ we can choose values that require an arbitrarily large number of bits to be represented or choose code values that can be represented with the minimum number of bits, given by Eq. (5.10). Now we show that the latter choice satisfies the sufficient condition for optimality.

To begin, we must consider that there is some overhead in a compressed file, which may include

- Extra bits required for saving \hat{v} with an integer number of bytes.
- A fixed or variable number of bits representing the number of symbols coded.
- Information about the probabilities (\mathbf{p} or \mathbf{c}).

Assuming that the total overhead is a positive number σ bits, we conclude from (5.10) that the number of bits per symbol used for coding a sequence S should be bounded by

$$B_S \leq \frac{\sigma - \log_2(l_N)}{N} \text{ bits/symbol.} \quad (5.21)$$

It follows from (5.9) that

$$l_N = \prod_{k=1}^N p(s_k), \quad (5.22)$$

and thus

$$B_S \leq \frac{\sigma - \sum_{k=1}^N \log_2 p(s_k)}{N} \text{ bits/symbol.} \quad (5.23)$$

Defining $E\{\cdot\}$ as the expected value operator, the expected number of bits per symbol is

$$\begin{aligned} \bar{B} = E\{B_S\} &\leq \frac{\sigma - \sum_{k=1}^N E\{\log_2 p(s_k)\}}{N} = \frac{\sigma - \sum_{k=1}^N \sum_{m=0}^{M-1} p(m) \log_2 p(m)}{N} \\ &\leq H(\Omega) + \frac{\sigma}{N}. \end{aligned} \quad (5.24)$$

Since the average number of bits per symbol cannot be smaller than the entropy, we have

$$H(\Omega) \leq \bar{B} \leq H(\Omega) + \frac{\sigma}{N}, \quad (5.25)$$

and it follows that

$$\lim_{N \rightarrow \infty} \{\bar{B}\} = H(\Omega), \quad (5.26)$$

which means that arithmetic coding indeed achieves optimal compression performance.

At this point we may ask why arithmetic coding creates intervals, instead of single code values. The answer lies in the fact that arithmetic coding is optimal not only for binary output—but rather for any output alphabet. In the final interval we find the different code values that are optimal for each output alphabet. Here is an example of use with non-binary outputs.

Example 5.5. Consider transmitting the data sequence of Example 5.3 using a communications system that conveys information using three levels, $\{-V, 0, +V\}$ (actually used in radio remote controls). Arithmetic coding with ternary output can simultaneously compresses the data and convert it to the proper transmission format.

The generalization of (5.10) for a D -symbol output alphabet is

$$B_{\min}(l_N, D) = \lceil -\log_D(l_N) \rceil \text{ symbols.} \quad (5.27)$$

Thus, using the results in Table 5.2, we conclude that we need $\lceil -\log_3(0.0002) \rceil = 8$ ternary symbols. We later show how to use standard arithmetic coding to find that the shortest ternary representation is $\hat{v}_3 = 0.20200111_3 \approx 0.742722146$, which means that the sequence $S = \{2, 1, 0, 0, 1, 3\}$ can be transmitted as the sequence of electrical signals $\{+V, 0, +V, 0, 0, -V, -V, -V\}$.

5.2.5 Arithmetic Coding Properties

5.2.5.1 Dynamic Sources

In Section 5.2.1 we assume that the data source Ω is stationary, so we have one set of symbol probabilities for encoding and decoding all symbols in the data sequence S . Now, with an understanding of the coding process, we generalize it for situations where the probabilities change

for each symbol coded, i.e., the k th symbol in the data sequence S is a random variable with probabilities \mathbf{p}_k and distribution \mathbf{c}_k .

The only required change in the arithmetic coding process is that instead of using (5.9) for interval updating, we should use

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c_k(s_k)l_{k-1}, p_k(s_k)l_{k-1}\rangle, \quad k = 1, 2, \dots, N. \quad (5.28)$$

To understand the changes in the decoding process, remember that the process of working with updated code values is equivalent to “erasing” all information about past symbols and decoding in the $[0, 1)$ interval. Thus, the decoder only has to use the right set of probabilities for that symbol to decode it correctly. The required changes to (5.16) and (5.17) yield

$$\hat{s}_k(\hat{v}) = \left\{ s : c_k(s) \leq \frac{\hat{v} - b_{k-1}}{l_{k-1}} < c_k(s+1) \right\}, \quad k = 1, 2, \dots, N, \quad (5.29)$$

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + c_k(\hat{s}_k(\hat{v}))l_{k-1}, p_k(\hat{s}_k(\hat{v}))l_{k-1}\rangle, \quad k = 1, 2, \dots, N. \quad (5.30)$$

Note that the number of symbols used at each instant can change. Instead of having a single input alphabet with M symbols, we have a sequence of alphabet sizes $\{M_1, M_2, \dots, M_N\}$.

5.2.5.2 Encoder and Decoder Synchronized Decisions

In data compression an encoder can change its behavior (parameters, coding algorithm, etc.) while encoding a data sequence, as long as the decoder uses the same information and the same rules to change its behavior. In addition, these changes must be “synchronized,” not in time, but in relation to the sequence of data source symbols.

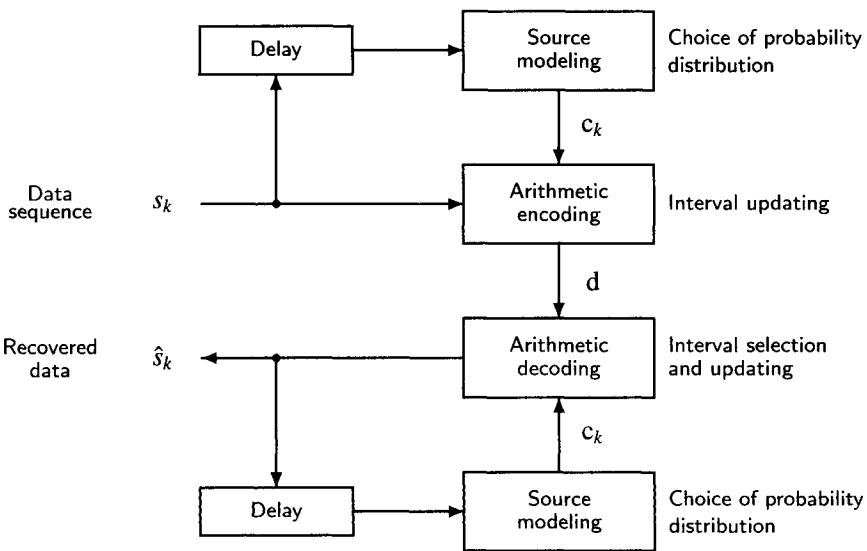
For instance, in Section 5.2.5.1, we assume that the encoder and decoder are synchronized in their use of varying sets of probabilities. Note that we do not have to assume that all the probabilities are available to the decoder when it starts decoding. The probability vectors can be updated with any rule based on symbol occurrences, as long as \mathbf{p}_k is computed from the data already available to the decoder, i.e., $\{\hat{s}_1, \hat{s}_2, \dots, \hat{s}_{k-1}\}$. This principle is used for adaptive coding, and it is covered in Section 5.3.2.

This concept of synchronization is essential for arithmetic coding because it involves a non-linear dynamic system (Fig. 5.3), and error accumulation leads to incorrect decoding, unless the encoder and decoder use *exactly* the same implementation (same precision, number of bits, rounding rules, equations, tables, etc.). In other words, we can make arithmetic coding work correctly even if the encoder makes coarse approximations, as long as the decoder makes exactly the same approximations. We have already seen an example of a choice based on numerical stability: Eq. (5.16) and (5.17) enable us to synchronize the encoder and decoder because they use the same interval updating rules used by (5.9), while (5.13) and (5.14) use a different recursion.

5.2.5.3 Separation of Coding and Source Modeling

There are many advantages for separating the source modeling (probabilities estimation) and the coding processes [14, 25, 29, 38, 45, 51, 53]. For example, it allows us to develop complex compression schemes without worrying about the details in the coding algorithm and/or use them with different coding methods and implementations.

Figure 5.5 shows how the two processes can be separated in a complete system for arithmetic encoding and decoding. The coding part is responsible only for updating the intervals; i.e., the arithmetic encoder implements recursion (5.28) and the arithmetic decoder implements (5.29) and (5.30). The encoding/decoding processes use the probability distribution vectors as input, but do not change them in any manner. The source modeling part is responsible for choosing the distribution \mathbf{c}_k that is used to encode/decode symbol s_k . Figure 5.5 also shows that a delay of one

**FIGURE 5.5**

Separation of coding and source modeling tasks. Arithmetic encoding and decoding process intervals, while source modeling chooses the probability distribution for each data symbol.

data symbol before the source-modeling block guarantees that encoder and decoder use the same information to update c_k .

Arithmetic coding simplifies considerably the implementation of systems like Fig. 5.5 because the vector c_k is used directly for coding. With Huffman coding, changes in probabilities require recomputing the optimal code or using complex code updating techniques [9, 24, 26].

5.2.5.4 Interval Rescaling

Figure 5.4 shows graphically one important property of arithmetic coding: The actual intervals used during coding depend on the initial interval and the previously coded data, but the proportions within subdivided interval do not. For example, if we change the initial interval to $\Phi_0 = [1, 2) = [1, 3)$ and apply (5.9), the coding process remains the same, except that all intervals are scaled by a factor of 2 and shifted by 1.

We can also apply rescaling in the middle of the coding process. Suppose that at a certain stage m we change the interval according to

$$b'_m = \gamma (b_m - \delta), \quad l'_m = \gamma l_m, \quad (5.31)$$

and continue the coding process normally (using (5.9) or (5.28)). When we finish coding we obtain the interval $\Phi'_N(S) = [b'_N, l'_N)$ and the corresponding code value v' . We can use the following equations to recover the interval and code value that we would have obtained without rescaling:

$$b_N = \frac{b'_N}{\gamma} + \delta, \quad l_N = \frac{l'_N}{\gamma}, \quad \hat{v} = \frac{v'}{\gamma} + \delta. \quad (5.32)$$

The decoder needs the original code value \hat{v} to start recovering the data symbols. It should also rescale the interval at stage m and thus needs to know m, δ, γ . Furthermore, when it scales

the interval using (5.31), it must scale the code value as well, using

$$v' = \gamma (\hat{v} - \delta). \quad (5.33)$$

We can generalize the results above to rescaling at stages $m \leq n \leq \dots \leq p$. In general, the scaling process, including the scaling of the code values, is

$$\begin{aligned} b'_m &= \gamma_1 (b_m - \delta_1), & l'_m &= \gamma_1 l_m, & v' &= \gamma_1 (\hat{v} - \delta_1), \\ b''_n &= \gamma_2 (b'_n - \delta_2), & l''_n &= \gamma_2 l'_n, & v'' &= \gamma_2 (v' - \delta_2), \\ \vdots & & \vdots & & \vdots & \\ b_p^{(T)} &= \gamma_T (b_p^{(T-1)} - \delta_T), & l_p^{(T)} &= \gamma_T l_p^{(T-1)}, & v^{(T)} &= \gamma_T (v^{(T-1)} - \delta_T). \end{aligned} \quad (5.34)$$

At the end of the coding process we have interval $\Phi_N(S) = |\bar{b}_N, \bar{l}_N\rangle$ and code value \bar{v} . We recover original values using

$$\Phi_N(S) = |b_N, l_N\rangle = \left| \delta_1 + \frac{1}{\gamma_1} \left(\delta_2 + \frac{1}{\gamma_2} \left(\delta_3 + \frac{1}{\gamma_3} \left(\dots \left(\delta_T + \frac{\bar{b}_N}{\gamma_T} \right) \right) \right) \right), \frac{\bar{l}_N}{\prod_{i=1}^T \gamma_i} \right\rangle, \quad (5.35)$$

and

$$\hat{v} = \delta_1 + \frac{1}{\gamma_1} \left(\delta_2 + \frac{1}{\gamma_2} \left(\delta_3 + \frac{1}{\gamma_3} \left(\dots \left(\delta_T + \frac{\bar{v}}{\gamma_T} \right) \right) \right) \right). \quad (5.36)$$

These equations may look awfully complicated, but in some special cases they are quite easy to use. For instance, in the next section we consider the cases when $\delta_i \in \{0, 1/2\}$ and $\gamma_i \equiv 2$ and show the connection between δ_i and the binary representation of b_N and \hat{v} . The next example shows another simple application of interval rescaling.

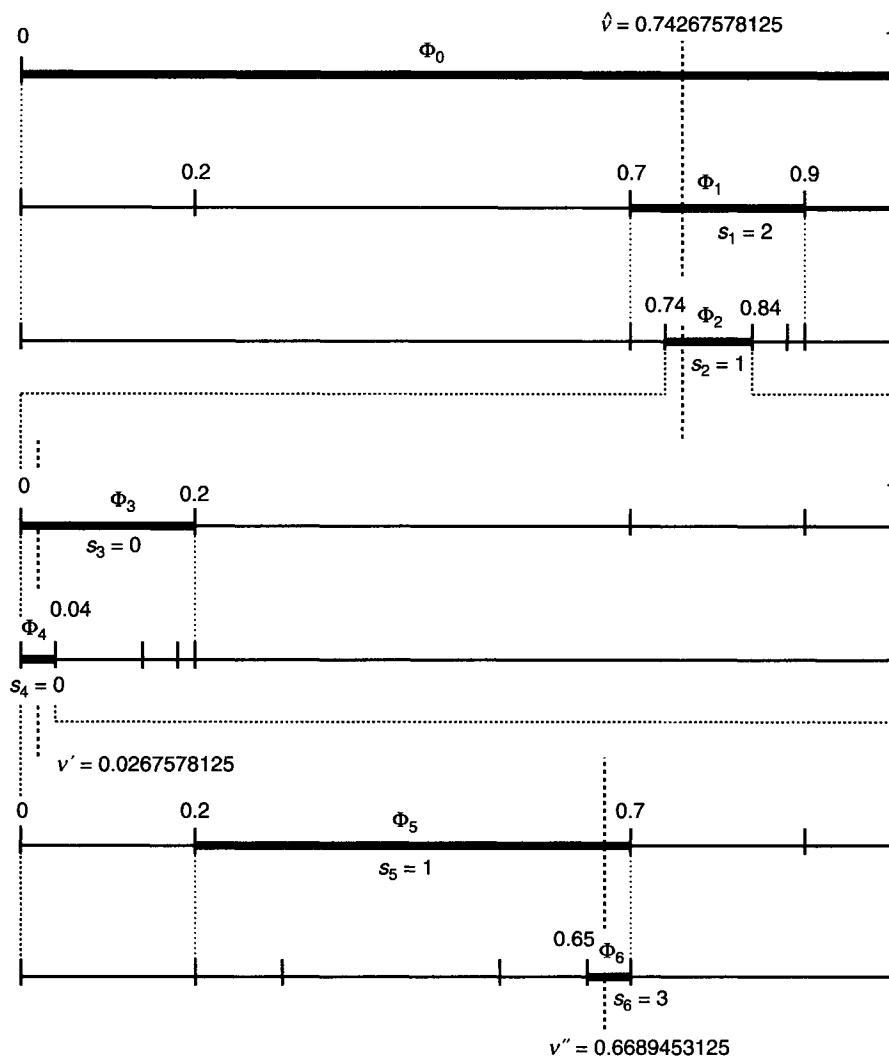
Example 5.6. Figure 5.6 shows rescaling applied to Example 5.3. It is very similar to Fig. 5.4, but instead of having just an enlarged view of small intervals, in Fig. 5.6 the intervals also change. The rescaling parameters $\delta_1 = 0.74$ and $\gamma_1 = 10$ are used after two symbols are coded and $\delta_2 = 0$ and $\gamma_2 = 25$ are used after two more symbols are coded. The final interval is $\Phi_6(S) = [0.65, 0.05]$, which corresponds to

$$\Phi_6(S) = \left[0.74 + \frac{1}{10} \left(\frac{0.65}{25} \right), \frac{0.05}{10 \times 25} \right] = [0.7426, 0.0002],$$

and which is exactly the interval obtained in Example 5.3.

5.2.5.5 Approximate Arithmetic

To understand how arithmetic coding can be implemented with fixed-precision we should note that the requirements for addition and for multiplication are quite different. We show that if we are willing to lose some compression efficiency, then we do not need exact multiplications. We use the double brackets $([\cdot])$ around a multiplication to indicate that it is an approximation; i.e., $[[\alpha \cdot \beta]] \approx \alpha \cdot \beta$. We define truncation as any approximation such that $[[\alpha \cdot \beta]] \leq \alpha \cdot \beta$. The approximation we are considering here can be rounding or truncation to any precision. The following example shows an alternative way to interpret inexact multiplications.

**FIGURE 5.6**

Graphical representation of the arithmetic coding process of Example 5.3 (Fig. 5.4) using numerical rescaling. Note that the code value changes each time the intervals are rescaled.

Example 5.7. We can see in Fig. 5.3 that the arithmetic coding multiplications always occur with data from the source model—the probability p and the cumulative distribution c . Suppose we have $l = 0.04$, $c = 0.317$, and $p = 0.123$, with

$$\begin{aligned} l \times c &= 0.04 \times 0.317 = 0.01268, \\ l \times p &= 0.04 \times 0.123 = 0.00492. \end{aligned}$$

Instead of using exact multiplication we can use an approximation (e.g., with table look-up and short registers) such that

$$\begin{aligned} [[l \times c]] &= [[0.04 \times 0.317]] = 0.012, \\ [[l \times p]] &= [[0.04 \times 0.123]] = 0.0048. \end{aligned}$$

Now, suppose that instead of using p and c , we had used another model, with $c' = 0.3$ and $p' = 0.12$. We would have obtained

$$l \times c' = 0.04 \times 0.3 = 0.012,$$

$$l \times p' = 0.04 \times 0.12 = 0.0048,$$

which are exactly the results with approximate multiplications. This shows that inexact multiplications are mathematically equivalent to making approximations in the source model and then using exact multiplications.

What we have seen in this example is that whatever the approximation used for the multiplications we can always assume that exact multiplications occur all the time, but with inexact distributions. We do not have to worry about the exact distribution values as long as the decoder is synchronized with the encoder, i.e., if the decoder is making exactly the same approximations as the encoder, then the encoder and decoder distributions must be identical (just like having dynamic sources, as explained in Section 5.2.5.1).

The version of (5.9) with inexact multiplications is

$$\Phi_k(S) = |b_k, l_k\rangle = |b_{k-1} + [[c(s_k) \cdot l_{k-1}]], [[p(s_k) \cdot l_{k-1}]]\rangle, \quad k = 1, 2, \dots, N. \quad (5.37)$$

We must also replace (5.16) and (5.17) with

$$\hat{s}_k(\hat{v}) = \{s : b_{k-1} + [[c(s) \cdot l_{k-1}]] \leq \hat{v} < b_{k-1} + [[c(s+1) \cdot l_{k-1}]]\}, \quad k = 1, 2, \dots, N, \quad (5.38)$$

$$\Phi_k(\hat{v}) = |b_k, l_k\rangle = |b_{k-1} + [[c(\hat{s}_k(\hat{v})) \cdot l_{k-1}]], [[p(\hat{s}_k(\hat{v})) \cdot l_{k-1}]]\rangle, \quad k = 1, 2, \dots, N. \quad (5.39)$$

In the next section we explain which conditions have to be satisfied by the approximate multiplications to have correct decoding.

In Eqs. (5.37) to (5.39) we have one type of approximation occurring from the multiplication of the interval length by the cumulative distribution and another approximation resulting from the multiplication of the interval length by the probability. If we want to use only one type of approximation and avoid multiplications between length and probability, we should update interval lengths according to

$$l_k = (b_{k-1} + [[c(s_k + 1) \cdot l_{k-1}]]) - (b_{k-1} + [[c(s_k) \cdot l_{k-1}]]) . \quad (5.40)$$

The price to pay for inexact arithmetic is degraded compression performance. Arithmetic coding is optimal only as long as the source model probabilities are equal to the true data symbol probabilities; any difference reduces the compression ratios.

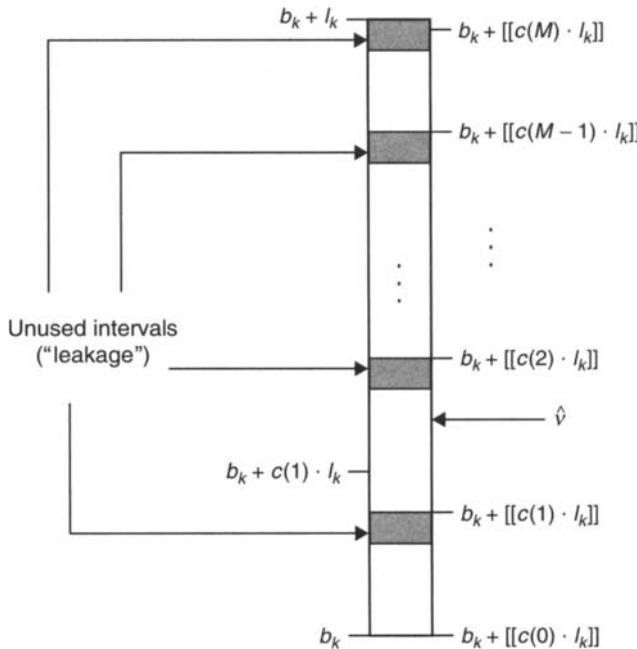
A quick analysis can give us an idea of how much can be lost. If we use a model with probability values \mathbf{p}' in a source with probabilities \mathbf{p} , the average loss in compression is

$$\Delta = \sum_{n=0}^{M-1} p(n) \log_2 \left[\frac{p(n)}{p'(n)} \right] \text{ bits/symbol.} \quad (5.41)$$

This formula is similar to the relative entropy [32], but in this case \mathbf{p}' represents the values that would result from the approximations, and it is possible to have $\sum_{n=0}^{M-1} p'(n) < 1$.

Assuming a relative multiplication error within ε , i.e.,

$$1 - \varepsilon \leq \frac{p(n)}{p'(n)} \leq 1 + \varepsilon, \quad (5.42)$$

**FIGURE 5.7**

Subdivision of a coding interval with approximate multiplications. Due to the fixed-precision arithmetic, we can only guarantee that all coding intervals are disjoint if we leave small regions between intervals unused for coding.

we have

$$\Delta \leq \sum_{n=0}^{M-1} p(n) \log_2(1 + \varepsilon) \approx \frac{\varepsilon}{\ln(2)} \approx 1.4\varepsilon \text{ bits/symbol.} \quad (5.43)$$

This is not a very tight bound, but it shows that if we can make multiplication accurately to, say, 4 digits, the loss in compression performance can be reasonably small.

5.2.5.6 Conditions for Correct Decoding

Figure 5.7 shows how an interval is subdivided when using inexact multiplications. In this figure we show that there can be a substantial difference between, say, $b_k + c(1) \cdot l_k$ and $b_k + [[c(1) \cdot l_k]]$, but this difference does not lead to decoding errors if the decoder uses the same approximation.

Decoding errors occur when condition (5.19) is not satisfied. Below we show the constraints that must be satisfied by approximations and analyze the three main causes of coding error to be avoided.

5.2.5.6.1 The Interval Length Must be Positive and Intervals Must be Disjoint The constraints that guarantee that the intervals do not collapse into a single point and that the interval length does not become larger than the allowed interval are, for $s = 0, 1, \dots, M - 1$,

$$0 < l_{k+1} = [[p(s) \cdot l_k]] \leq (b_k + [[c(s+1) \cdot l_k]]) - (b_k + [[c(s) \cdot l_k]]). \quad (5.44)$$

For example, if the approximations can create a situation in which $[[c(s+1) \cdot l_k]] < [[c(s) \cdot l_k]]$, there would be a non-empty intersection of subintervals assigned for $s+1$ and s , and decoder errors could occur whenever a code value belongs to the intersection.

If $[[c(s+1) \cdot l_k]] = [[c(s) \cdot l_k]]$, then the interval length collapses to zero and stays as such, independently of the symbols coded next. The interval length may become zero due to arithmetic underflow, when both l_k and $p(s) = c(s+1) - c(s)$ are very small. In Section 5.3.1 we show that interval rescaling is normally used to keep l_k within a certain range to avoid this problem, but we also must be sure that all symbol probabilities are larger than a minimum value defined by the arithmetic precision (see (5.63) and (5.76)).

Besides the conditions defined by (5.44), we also need to have

$$[[c(0) \cdot l_k]] \geq 0, \quad \text{and} \quad [[c(M) \cdot l_k]] \leq l_k. \quad (5.45)$$

These two condition are easier to satisfy because $c(0) \equiv 0$ and $c(M) \equiv 1$, and it is easy to make such multiplications exact.

5.2.5.6.2 Subintervals Must be Nested We must be sure that the accumulation of the approximation errors, as we continue coding symbols, does not move the interval base to a point outside all the previous intervals. With exact arithmetic, as we code new symbols, the interval base increases within the interval assigned to s_{k+1} , but it never crosses the boundary to the interval assigned to $s_{k+1} + 1$, i.e.,

$$b_{k+n} = b_k + \sum_{i=k}^{k+n-1} c(s_{i+1}) \cdot l_i < b_k + c(s_{k+1} + 1) \cdot l_k, \quad \text{for all } n \geq 0. \quad (5.46)$$

The equivalent condition for approximate arithmetic is that for every data sequence we must have

$$b_k + [[c(s_{k+1} + 1) \cdot l_k]] > b_k + [[c(s_{k+1}) \cdot l_k]] + \sum_{i=k+1}^{\infty} [[c(s_{i+1}) \cdot l_i]]. \quad (5.47)$$

To determine when (5.47) may be violated we have to assume some limits on the multiplication approximations. There should be a non-negative number ε such that

$$[[c(s_{i+1}) \cdot l_i]](1 - \varepsilon) < c(s_{i+1}) \cdot l_i. \quad (5.48)$$

We can combine (5.46), (5.47), and (5.48) to obtain

$$(1 - \varepsilon) \cdot l_k > \sum_{i=k+1}^{\infty} c(s_{i+1}) \cdot l_i, \quad (5.49)$$

which is equal to

$$1 - \varepsilon > c(s_{k+2}) + p(s_{k+3})(c(s_{k+3}) + p(s_{k+3})(c(s_{k+4}) + p(s_{k+4})(\dots))). \quad (5.50)$$

To find the maximum for the right-hand side of (5.50) we only have to consider the case $s_{k+2} = s_{k+3} = \dots = M - 1$ to find

$$1 - \varepsilon > c(M - 1) + p(M - 1)(c(M - 1) + p(M - 1)(c(M - 1) + p(M - 1)(\dots))), \quad (5.51)$$

which is equivalent to

$$1 - \varepsilon > c(M - 1) + p(M - 1). \quad (5.52)$$

But we know from (5.3) that by definition $c(M - 1) + p(M - 1) \equiv 1$! The answer to this contradiction lies in the fact that with exact arithmetic we would have equality in (5.46) only after an infinite number of symbols. With inexact arithmetic it is impossible to have semiopen

intervals that are fully used and match perfectly, so we need to take some extra precautions to be sure that (5.47) is always satisfied. What Eq. (5.52) tells us is that we solve the problem if we artificially decrease the interval range assigned for $p(M - 1)$. This is equivalent to setting aside small regions, indicated as gray areas in Fig. 5.7, that are not used for coding and serve as a “safety net.”

This extra space can be intentionally added, for example, by replacing (5.40) with

$$l_k = (b_{k-1} + [[c(s_k + 1) \cdot l_{k-1}]]) - (b_{k-1} + [[c(s_k) \cdot l_{k-1}]]) - \zeta \quad (5.53)$$

where $0 < \zeta \ll 1$ is chosen to guarantee correct coding and small compression loss.

The loss in compression caused by these unused subintervals is called “leakage” because a certain fraction of bits is “wasted” whenever a symbol is coded. This fraction is on average

$$\Delta_s = p(s) \log_2 \left(\frac{p(s)}{p'(s)} \right) \text{ bits}, \quad (5.54)$$

where $p(s)/p'(s) > 1$ is the ratio between the symbol probability and the size of interval minus the unused region. With reasonable precision, leakage can be made extremely small. For instance, if $p(s)/p'(s) = 1.001$ (low precision), then leakage is less than 0.0015 bits/symbol.

5.2.5.6.3 Inverse Arithmetic Operations Must Not Produce Error Accumulation Note that in (5.38) we define decoding assuming only the additions and multiplications used by the encoder. We could have used

$$\hat{s}_k(\hat{v}) = \left\{ s : c(s) \leq \left[\left[\frac{\hat{v} - b_{k-1}}{l_{k-1}} \right] \right] < c(s + 1) \right\}, \quad k = 1, 2, \dots, N. \quad (5.55)$$

However, this introduces approximate subtraction and division, which must be consistent with the encoder’s approximations. Here we cannot possibly cover all problems related to inverse operations, but we should say that the main point is to observe error accumulation.

For example, we can exploit the fact that in (5.16) decoding uses only the difference $\varpi_k \equiv \hat{v} - b_k$ and use the following recursions.

$$|\varpi_0, l_0\rangle = |\hat{v}, 1\rangle, \quad (5.56)$$

$$\hat{s}_k = \{s : [[c(s) \cdot l_{k-1}]] \leq \varpi_k < [[c(s + 1) \cdot l_{k-1}]]\}, \quad k = 1, 2, \dots, N. \quad (5.57)$$

$$|\varpi_k, l_k\rangle = |\varpi_{k-1} - [[c(\hat{s}_k) \cdot l_{k-1}]], [[p(\hat{s}_k) \cdot l_{k-1}]]\rangle, \quad k = 1, 2, \dots, N. \quad (5.58)$$

However, because we are using a sequence of subtractions in (5.58), this technique works with integer arithmetic implementations (see Appendix 5.1), but it may not work with floating-point implementations because of error accumulation.

5.3 IMPLEMENTATION

In this second part, we present the practical implementations of arithmetic coding. We show how to exploit all the arithmetic coding properties presented in the previous sections and develop a version that works with fixed-precision arithmetic. First, we explain how to implement binary extended-precision additions that exploit the arithmetic coding properties, including the carry propagation process. Next, we present complete encoding and decoding algorithms based on an efficient and simple form of interval rescaling. We provide the description for both floating-point and integer arithmetic and present some alternative ways of implementing the coding, including different scaling and carry propagation strategies. After covering the details of the coding process, we

study the symbol probability estimation problem and explain how to implement adaptive coding by integrating coding and source modeling. At the end, we analyze the computational complexity of arithmetic coding.

5.3.1 Coding with Fixed-Precision Arithmetic

Our first practical problem is that the number of digits (or bits) required to represent the interval length exactly grows when a symbol is coded. For example, if we had $p(0) = 0.99$ and we repeatedly code symbol 0, we would have

$$l_0 = 1, \quad l_1 = 0.99, \quad l_2 = 0.9801, \quad l_3 = 0.970299, \quad l_4 = 0.96059601, \dots$$

We solve this problem using the fact we do not need exact multiplications by the interval length (Section 5.2.5.5). Practical implementations use P -bit registers to store approximations of the mantissa of the interval length and the results of the multiplications. All bits with significance smaller than those in the register are assumed to be zero.

With the multiplication precision problem solved, we still have the problem of implementing the additions in (5.37) when there is a large difference between the magnitudes of the interval base and interval length. We show that rescaling solves the problem, simultaneously enabling exact addition and reducing loss of multiplication accuracy. For a binary output, we can use rescaling in the form of (5.31), with $\delta \in \{0, 0.5\}$ and $\gamma = 2$ whenever the length of the interval is below 0.5. Since the decoder needs to know the rescaling parameters, they are saved in the data buffer \mathbf{d} , using bits “0” or “1” to indicate whether $\delta = 0$ or $\delta = 0.5$. Special case $\delta = 1$ and $\gamma = 1$, corresponding to a carry in the binary representation, is explained later.

To simplify the notation we represent the rescaled intervals simply as $|b, l\rangle$ (no subscripts), and the rescaled code value as v :

$$\begin{aligned} l &= 2^{t(l_k)} l_k, \\ b &= \text{frac}(2^{t(l_k)} b_k) = 2^{t(l_k)} b_k - \lfloor 2^{t(l_k)} b_k \rfloor, \\ v &= \text{frac}(2^{t(l_k)} \hat{v}), \end{aligned} \tag{5.59}$$

where $\text{frac}(\cdot)$ is the fractional part of a number and

$$t(x) = \{n : 2^{-n-1} < x \leq 2^{-n}\} = \lfloor -\log_2(x) \rfloor. \tag{5.60}$$

Note that under these conditions we have $b \in [0, 1)$ and $l \in (0.5, 1]$, and for that reason the rescaling process is called *renormalization*. Of course, l , b , and v change with k , but this new notation is more convenient to represent variables in algorithm descriptions or computer programs.

The binary representations of the interval base and length have the structure

$$\begin{array}{cccc} l_k = 0.0000 \dots 00 & 0000 \dots 00 & \overbrace{1aaa \dots aa}^{(L=2^P l)} & 000000 \dots 2 \\ b_k = \underbrace{0.aaaa \dots aa}_{\text{settled}} & \underbrace{0111 \dots 11}_{\text{outstanding}} & \underbrace{aaaa \dots aa}_{(B=2^P b) \text{ active}} & \underbrace{000000 \dots 2}_{\text{trailing zeros}} \end{array}, \tag{5.61}$$

where symbol a represents an arbitrary bit value.

We can see in (5.61) that there is “window” of P *active* bits, forming integers L and B , corresponding to the non-zero bits of l_k and the renormalized length l . Because the value of l

ALGORITHM 5.1**Function Arithmetic_Encoder ($N, S, M, \mathbf{c}, \mathbf{d}$)**

-
- | | |
|--|---|
| 1. set $\{ b \leftarrow 0; l \leftarrow 1;$
$t \leftarrow 0; \}$
2. for $k = 1$ to N do
2.1. Interval_Update (s_k, b, l, M, \mathbf{c});
2.2. if $b \geq 1$, then
2.2.1. set $\{ b \leftarrow b - 1;$
Propagate_Carry (t, \mathbf{d}); $\}$
2.3. if $l \leq 0.5$, then
2.3.1. Encoder_Renormalization (b, l, t, \mathbf{d});
3. Code_Value_Selection (b, t, \mathbf{d});
4. return t . | <ul style="list-style-type: none"> ★ Initialize interval ★ and bit counter ★ Encode N data symbols ★ Update interval according to symbol ★ Check for carry ★ Shift interval base ★ Propagate carry on buffer ★ If interval is small enough,
★ then renormalize interval ★ Choose final code value ★ Return number of code bits |
|--|---|
-

is truncated to P -bit precision, there is a set of trailing zeros that does not affect the additions. The bits to the left of the active bits are those that had been saved in the data buffer \mathbf{d} during renormalization, and they are divided in two sets.

The first set to the left is the set of *outstanding* bits: those that can be changed due to a carry from the active bits when new symbols are encoded. The second is the set of bits that have been *settled*; i.e., they stay constant until the end of the encoding process. This happens because intervals are nested; i.e., the code value cannot exceed

$$b_{k+n} < b_k + l_k \leq \underbrace{0.aaaa\dots aa}_{\text{settled}} \quad \underbrace{1000\dots 00}_{\text{changed by carry}} \quad \underbrace{aaaa\dots aa}_{\text{active}} \quad 00000\dots_2. \quad (5.62)$$

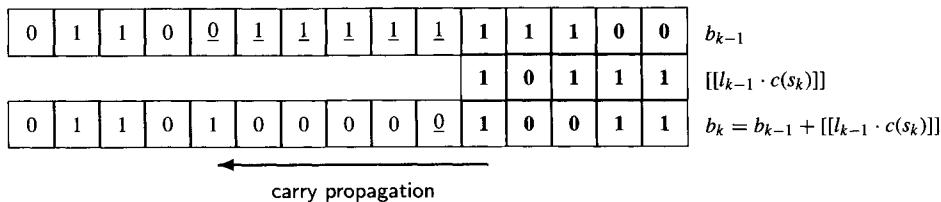
This equation shows that only the outstanding bits may change due to a carry from the active bits. Furthermore, inequality (5.62) also shows that there can be only one carry that would change these bits. If there is a carry, or when it is found that there can be no carry, these bits become settled. For that reason, the set of outstanding bits always start with 0 and is possibly followed only by 1's. As new symbols are encoded, all sets move to the right.

5.3.1.1 Implementation with Buffer Carries

Combining all we have seen, we can present an encoding algorithm that works with fixed-precision arithmetic. Algorithm 5.1 shows a function `Arithmetic_Encoder` to encode a sequence S of N data symbols, following the notation of Section 5.2.1. This algorithm is very similar to the encoding process that we used in Section 5.2.3, but with a renormalization stage after each time a symbol is coded and the settled and outstanding bits being saved in the buffer \mathbf{d} . The function returns the number of bits used to compress S .

In Algorithm 5.1, Step 1 sets the initial interval equal to $[0, 1)$ and initializes the bit counter t to zero. Note that we use curly braces ($\{ \}$) to enclose a set of assignments and use symbol “★” before comments. In Step 2, we have the sequential repetition of interval resizing and renormalizations. Immediately after updating the interval we find out if there is a carry, i.e., if $b \geq 1$, and next we check if further renormalization is necessary. The encoding process finishes in Step 3, when the final code value v that minimizes the number of code bits is chosen. In all our algorithms, we assume that functions receive references; i.e., variables can be changed inside the function called. Below we describe each of the functions used by Algorithm 5.1.

There are many mathematically equivalent ways of updating the interval $|b, l\rangle$. We do not need to have both vectors \mathbf{p} and \mathbf{c} stored to use (5.9). In Algorithm 5.2 we use (5.40) to update length

**FIGURE 5.8**

Carry propagation process. The active bits are indicated in boldface type, outstanding bits are underlined, and left-most bits are settled.

ALGORITHM 5.2

Procedure Interval_Update (s, b, l, M, c)

1. if $s = M - 1$,
then set $y \leftarrow b + l$;
else set $y \leftarrow b + l \cdot c(s + 1)$;
 2. set { $b \leftarrow b + l \cdot c(s)$;
 $l \leftarrow y - b$; }
 3. return.
- ★ Special case for last symbol
 - ★ end of interval
 - ★ base of next subinterval
 - ★ Update interval base
 - ★ Update interval length as difference
-

ALGORITHM 5.3

Procedure Propagate_Carry (t, d)

1. set $n \leftarrow t$;
 2. while $d(n) = 1$ do
 - 2.1. set { $d(n) \leftarrow 0$;
 $n \leftarrow n - 1$; }
 3. set $d(n) \leftarrow 1$;
 4. return.
- ★ Initialize pointer to last outstanding bit
 - ★ While carry propagation
 - ★ complement outstanding 1-bit and
 - ★ move to previous bit
 - ★ Complement outstanding 0-bit
-

as a difference, and we avoid multiplication for the last symbol ($s = M - 1$), since it is more efficient to do the same at the decoder. To simplify notation, we do not use double brackets to indicate inexact multiplications, but it should be clear that here all numbers represent the content of CPU registers.

In Step 2.2.1 of Algorithm 5.1, the function to propagate the carry in the buffer \mathbf{d} is called, changing bits that have been added to \mathbf{d} previously, and we shift the interval to have $b < 1$. Figure 5.8 shows the carry propagation process. Active bits are shown in boldface type and outstanding bits are underlined. Whenever there is a carry, starting from the most recent bits added to buffer \mathbf{d} , we complement all bits until the first 0 bit is complemented, as in Algorithm 5.3.

Algorithm 5.4 implements interval renormalization, where we test if active bits became outstanding or settled. While the interval length is smaller than 0.5, the interval is rescaled by a factor of 2, and a bit is added to the bit buffer \mathbf{d} .

The final encoding stage is the selection of the code value (Algorithm 5.5). We use basically the same process explained at the end of Section 5.2.3.1, but here we choose a code value belonging

ALGORITHM 5.4**Procedure Encoder_Renormalization (b, l, t, \mathbf{d})**

1. while $l \leq 0.5$ do
 - 1.1. set $\{ t \leftarrow t + 1;$
 $l \leftarrow 2l; \}$
 - 1.2. if $b \geq 0.5$,
 then set $\{ d(t) \leftarrow 1;$
 $b \leftarrow 2(b - 0.5); \}$;
 - else set $\{ d(t) \leftarrow 0;$
 $b \leftarrow 2b; \}$
 2. return.
-

★ Renormalization loop

★ Increment bit counter and

★ Scale interval length

★ Test most significant bit of interval base

★ Output bit 1

★ Shift and scale interval base

★ Output bit 0

★ Scale interval base

ALGORITHM 5.5**Procedure Code_Value_Selection (b, t, \mathbf{d})**

1. set $t \leftarrow t + 1;$
 2. if $b \leq 0.5$,
 then set $d(t) \leftarrow 1;$
 else set $\{ d(t) \leftarrow 0;$
 $\text{Propagate_Carry}(t - 1, \mathbf{d}); \}$
 3. return.
-

★ Increment bit counter

★ Renormalize code value selection

★ Choose $v = 0.5$: output bit 1★ Choose $v = 1.0$: output bit 0 and

★ Propagate carry

ALGORITHM 5.6**Procedure Arithmetic_Decoder ($N, M, \mathbf{c}, \mathbf{d}, \hat{S}$)**

1. set $\{ b \leftarrow 0; l \leftarrow 1;$
 $v = \sum_{n=1}^P 2^{-n} d(n);$
 $t \leftarrow P; \}$
 2. for $k = 1$ to N do
 - 2.1. $\hat{s}_k = \text{Interval_Selection}(v, b, l, M, \mathbf{c});$
 - 2.2. if $b \geq 1$, then
 - 2.2.1. set $\{ b \leftarrow b - 1;$
 $v \leftarrow v - 1; \}$
 - 2.3. if $l \leq 0.5$, then
 - 2.3.1. $\text{Decoder_Renormalization}(v, b, l, t, \mathbf{d});$
 3. return.
-

★ Initialize interval

★ Read P bits of code value

★ Initialize bit counter

★ Decode N data symbols

★ Decode symbol and update interval

★ Check for “overflow”

★ Shift interval base

★ Shift code value

★ If interval is small enough,

★ then normalize interval

to the rescaled interval. Our choice is made easy because we know that, after renormalization, we always have $0.5 < l \leq 1$ (see (5.59)), meaning that we only need an extra bit to define the final code value. In other word, all bits that define the code value are already in buffer \mathbf{d} , and we need to choose only one more bit. The only two choices to consider in the rescaled interval are $v = 0.5$ or $v = 1$.

The decoding procedure, shown in Algorithm 5.6, gets as input the number of compressed symbols, N , the number of data symbols, M , and their cumulative distribution \mathbf{c} , and the array

ALGORITHM 5.7**Function Interval_Selection (v, b, l, M, c)**

-
1. set { $s \leftarrow M - 1$;
 $x \leftarrow b + l \cdot c(M - 1)$;
 $y \leftarrow b + l$; }
 2. while $x > v$ do
 - 2.1. set { $s \leftarrow s - 1$;
 $y \leftarrow x$;
 $x \leftarrow b + l \cdot c(s)$; }
 3. set { $b \leftarrow x$;
 $l \leftarrow y - b$; }
 4. return s .
-

- * Start search from last symbol
- * Base of search interval
- * End of search interval
- * Sequential search for correct interval
- * Decrement symbol by one
- * Move interval end
- * Compute new interval base
- * Update interval base
- * Update interval length as difference

ALGORITHM 5.8**Procedure Decoder_Renormalization (v, b, l, t, d)**

-
1. while $l \leq 0.5$ do
 - 1.1. if $b \geq 0.5$,
 then set { $b \leftarrow 2(b - 0.5)$;
 $v \leftarrow 2(v - 0.5)$; };
 - else set { $b \leftarrow 2b$;
 $v \leftarrow 2v$; }
 - 1.2. set { $t \leftarrow t + 1$;
 $v \leftarrow v + 2^{-P}d(t)$;
 $l \leftarrow 2l$; }
 2. return.
-

- * Renormalization loop
- * Remove most significant bit
- * Shift and scale interval base
- * Shift and scale code value
- * Scale interval base
- * Scale code value
- * Increment bit counter
- * Set least significant bit of code value
- * Scale interval length

with the compressed data bits, d . Its output is the recovered data sequence \hat{S} . The decoder must keep the P -bit register with the code value updated, so it will read P extra bits at the end of d . We assume that this can be done without problems and that those bits had been set to zero.

The interval selection is basically an implementation of (5.38): We want to find the subinterval that contains the code value v . The implementation that we show in Algorithm 5.7 has one small shortcut: It combines the symbol decoding with interval updating (5.39) in a single function. We do a sequential search, starting from the last symbol ($s = M - 1$), because we assume that symbols are sorted by increasing probability. The advantages of sorting symbols and more efficient searches are explained in Section 5.3.

In Algorithm 5.7 we use only arithmetic operations that are exactly equal to those used by the encoder. This way we can easily guarantee that the encoder and decoder approximations are exactly the same. Several simplifications can be used to reduce the number of arithmetic operations (see Appendix 5.1). However, we must be careful with divisions because a single division can take more time than several multiplications and additions.

The renormalization in Algorithm 5.8 is similar to Algorithm 5.4, and in fact all its decisions (comparisons) are meant to be based on exactly the same values used by the encoder. However, it also needs to rescale the code value in the same manner as the interval base (compare (5.35) and (5.36)), and it reads its least significant bit (with value 2^{-P}).

Table 5.3 Results of Arithmetic Encoding and Decoding, with Renormalization, Applied to Source and Data Sequence of Example 5.3

Event	Scaled Base b	Scaled Length l	Bit Buffer d	Scaled Code Value v	Normalized Code Value $(v - b)/l$
—	0	1		0.74267578125	0.74267578125
$s = 2$	0.7	0.2		0.74267578125	
$\delta = 0.5$	0.4	0.4	1	0.4853515625	
$\delta = 0$	0.8	0.8	10	0.970703125	0.21337890625
$s = 1$	0.96	0.4	10	0.970703125	
$\delta = 0.5$	0.92	0.8	101	0.94140625	0.0267578125
$s = 0$	0.92	0.16	101	0.94140625	
$\delta = 0.5$	0.84	0.32	1011	0.8828125	
$\delta = 0.5$	0.68	0.64	10111	0.765625	0.1337890625
$s = 0$	0.68	0.128	10111	0.765625	
$\delta = 0.5$	0.36	0.256	101111	0.53125	
$\delta = 0$	0.72	0.512	1011110	1.0625	0.6689453125
$s = 1$	0.8224	0.256	1011110	1.0625	
$\delta = 0.5$	0.6448	0.512	10111101	1.125	0.937890625
$s = 3$	1.1056	0.0512	10111101	1.125	
$\delta = 1$	0.1056	0.0512	10111110	0.125	
$\delta = 0$	0.2112	0.1024	101111100	0.25	
$\delta = 0$	0.4224	0.2048	1011111000	0.5	
$\delta = 0$	0.8448	0.4096	10111110000	1	
$\delta = 0.5$	0.6896	0.8192	101111100001	1	
$v = 1$	1	—	101111100001	—	
$\delta = 1$	0	—	101111100010	—	
$\delta = 0$	0	—	1011111000100	—	

Note: Final code value is $\hat{v} = 0.1011111000100_2 = 0.74267578125$.

Example 5.8. We applied Algorithms 5.1 to 5.8 to the source and data sequence of Example 5.3, and the results are shown in Table 5.3. The first column shows what event caused the change in the interval. If a new symbol σ is coded, then we show it as $s = \sigma$ and show the results of Algorithm 5.2. We indicate the interval changes during renormalization (Algorithm 5.4) by showing the value of δ used for rescaling, according to (5.31).

Comparing these results with those in Table 5.2, we see how renormalization keeps all numerical values within a range that maximizes numerical accuracy. In fact, the results in Table 5.3 are exact and can be shown with a few significant digits. Table 5.3 also shows the decoder's updated code value. Again, these results are exact and agree with the results shown in Table 5.2. The third column in Table 5.3 shows the contents of the bit buffer d and the bits that are added to this buffer every time the interval is rescaled. Note that carry propagation occurs twice: when $s = 3$ and when the final code value $v = 1$ is chosen by Algorithm 5.5.

5.3.1.2 Implementation with Integer Arithmetic

Even though we use real numbers to describe the principles of arithmetic coding, most practical implementations use only integer arithmetic. The adaptation is quite simple, as we just have to assume that the P -bit integers contain the fractional part of the real numbers, with the following adaptations. (Appendix 5.1 has the details.)

- Define $B = 2^P b$, $L = 2^P l$, $V = 2^P v$, and $C(s) = 2^P c(s)$. Products can be computed with $2P$ bits, and the P least-significant bits are discarded. For example, when updating the interval length we compute $L \leftarrow \lfloor L \cdot [C(s+1) - C(s)] \cdot 2^{-P} \rfloor$. The length value $l = 1$ cannot be represented in this form, but this is not a real problem. We only need to initialize the scaled length with $L \leftarrow 2^P - 1$ and apply renormalization only when $l < 0.5$ (strict inequality).
- The carry condition $b \geq 1$ is equivalent to $B \geq 2^P$, which can mean integer overflow. It can be detected accessing the CPU's carry flag or, equivalently, checking when the value of B decreases.
- Since $l > 0$ we can work with a scaled length equal to $L' = 2^P l - 1$. This way we can represent the value $l = 1$ and have some extra precision if P is small. On the other hand, updating the length using $L' \leftarrow \lfloor (L' + 1) \cdot [C(s+1) - C(s)] \cdot 2^{-P} - 1 \rfloor$ requires two more additions.

When multiplication is computed with $2P$ bits, we can determine what is the smallest allowable probability to avoid length underflow. Since renormalization guarantees that $L \geq 2^{P-1}$, we have satisfied (5.44) if

$$\lfloor [C(s+1) - C(s)] 2^{-P} L \rfloor \geq \lfloor [C(s+1) - C(s)] 2^{-1} \rfloor \geq 1 \Rightarrow C(s+1) - C(s) \geq 2, \quad (5.63)$$

meaning that the minimum probability supported by such implementations with P -bit integers is $p(s) \geq 2^{1-P}$. Different renormalizations change this value, according to the smallest interval length allowed after renormalization.

Example 5.9. Table 5.4 shows the results of an 8-bit register implementation ($P = 8$) applied to the data sequence and source used in Example 5.3. The table shows the interval $|b_k, l_k\rangle$ in base-2 notation to make clear that even though we use 8-bit arithmetic, we are actually implementing *exact* additions. Following the conventions in (5.61) and Fig. 5.8, we used boldface type to indicate active bits and underlined numbers for outstanding bits. We also show the approximate results of the multiplications $l_{k-1} \cdot c(s_k)$. The last column shows the binary contents of the registers with active bits. We used $L' = 2^P l - 1$ to represent the length. The results are also shown in decimal notation so that they can be compared with the exact results in Table 5.2. Note that the approximations change the code value after only 7 bits, but the number of bits required to represent the final code value is still 13 bits.

5.3.1.3 Efficient Output

Implementations with short registers (as in Example 5.9) require renormalizing the intervals as soon as possible to avoid losing accuracy and compression efficacy. We can see in Table 5.3 that, as a consequence, intervals may be rescaled many times whenever a symbol is coded. Even though rescaling can be done with bit-shifts instead of multiplications, this process still requires many CPU cycles (see Section 5.3.3).

If we use longer registers (e.g., 16 bits or more), we can increase efficiency significantly by moving more than 1 bit to the output buffer \mathbf{d} whenever rescaling. This process is equivalent to have an encoder output alphabet with D symbols, where D is a power of 2. For example, moving groups of 1, 2, 4, or 8 bits at a time, corresponds to output alphabets with $D = 2, 4, 16$, and 256 symbols, respectively. Carry propagation and the use of the output buffer \mathbf{d} also become more efficient with larger D .

It is not difficult to modify the algorithms in Section 5.3.1.1 for a D -symbol output. Renormalization is the most important change. The rescaling parameters defined by (5.31) are such that $\delta \in \{0, 1/D, 2/D, \dots, 1\}$ and $\gamma \equiv D$. Again, $\delta = 1$ corresponds to a carry. Algorithm 5.9 has the required changes to Algorithm 5.4, and the corresponding changes in the decoder renormalization

Table 5.4 Results of Arithmetic Encoding and Decoding for Example 5.3 Using 8-bit Precision for the Arithmetic Operations

Input s_k	Encoder Data	Binary Representation	Decimal Representation	8-bit Registers
—	b_0	0.00000000 ₂	0	00000000
	l_0	1.00000000 ₂	1	11111111
2	$l_0 \cdot c(2)$	0.10110011 ₂	0.69921875	10110011
	b_1	0.1011001100 ₂	0.69921875	11001100
	l_1	0.0011001100 ₂	0.19921875	11001011
1	$l_1 \cdot c(1)$	0.0000101000 ₂	0.0390625	00101000
	b_2	0.101111010000 ₂	0.73828125	11101000
	l_2	0.000011001100 ₂	0.099609375	11001011
0	$l_2 \cdot c(0)$	0.000000000000 ₂	0	00000000
	b_3	0.10111101000000 ₂	0.73828125	10100000
	l_3	0.00000010100000 ₂	0.01953125	10011111
0	$l_3 \cdot c(0)$	0.00000000000000 ₂	0	00000000
	b_4	0.1011110100000000 ₂	0.73828125	00000000
	l_4	0.000000000111110000 ₂	0.0037841796875	11110111
1	$l_4 \cdot c(1)$	0.00000000000110001 ₂	0.0007476806640625	00110001
	b_5	0.10111101001100010 ₂	0.7390289306640625	01100010
	l_5	0.000000000011111000 ₂	0.00189208984375	11110111
3	$l_5 \cdot c(3)$	0.000000000011011110 ₂	0.0016937255859375	11011110
	b_6	0.101111011010000000 ₂	0.74072265625	00000000
—	l_6	0.0000000000000011001000 ₂	0.00019073486328125	11000111
	\hat{v}	0.1011110110101 ₂	0.7408447265625	—

ALGORITHM 5.9

Procedure Encoder_Renormalization (b, l, t, d)

1. while $l \leq 1/D$ do
 - 1.1. set { $t \leftarrow t + 1;$ }
 - $d(t) \leftarrow \lfloor D \cdot b \rfloor;$
 - $b \leftarrow D \cdot b - d(t);$
 - $l \leftarrow D \cdot l;$ }
 2. return.
- ★ Renormalization loop
 ★ Increment symbol counter
 ★ Output symbol from most significant bits
 ★ Update interval base
 ★ Scale interval length

are shown in Algorithm 5.10. In Appendix 5.1 we have an integer-based implementation with D -symbol output, with the carry propagation in Algorithm 5.25.

We also need to change the equations that define the normalized intervals, (5.59) and (5.60), to

$$\begin{aligned}
 l &= D^{t(l_k)} l_k, \\
 b &= \text{frac}(D^{t(l_k)} b_k) = D^{t(l_k)} b_k - \lfloor D^{t(l_k)} b_k \rfloor, \\
 v &= \text{frac}(D^{t(l_k)} \hat{v}),
 \end{aligned} \tag{5.64}$$

Table 5.5 Results of Arithmetic Encoding and Decoding for Example 5.3, with Renormalization and a 16-symbol (hexadecimal) Output Alphabet

Event	Scaled Base b	Scaled Length l	Bit Buffer d	Scaled Code Value v
—	0	1		0.74267578125
$s = 2$	0.7	0.2		0.74267578125
$s = 1$	0.74	0.1		0.74267578125
$s = 0$	0.74	0.02		0.74267578125
$\delta = 11/16$	0.84	0.32	B	0.8828125
$s = 0$	0.84	0.064	B	0.8828125
$s = 1$	0.8528	0.032	B	0.8828125
$\delta = 13/16$	0.6448	0.512	BD	1.125
$s = 3$	1.1056	0.0512	BD	1.125
$\delta = 1$	0.1056	0.0512	BE	0.125
$\delta = 1/16$	0.6896	0.8192	BE1	1
$v = 1$	1	—	BE1	—
$\delta = 1$	0	—	BE2	—
$\delta = 0$	—	—	BE20	—

Note: Final code value is $\hat{v} = 0.\text{BE20}_{16} = 0.1011\ 1110\ 0010\ 0000_2 = 0.74267578125$.

ALGORITHM 5.10

Procedure Decoder_Renormalization (v, b, l, t, d)

1. while $l \leq 1/D$ do
 - 1.1. set { $t \leftarrow t + 1;$ }
 $a \leftarrow \lfloor D \cdot b \rfloor;$
 $b \leftarrow D \cdot b - a;$
 $v \leftarrow D \cdot v - a + D^{-P}d(t);$
 $l \leftarrow D \cdot l;$ }
2. return.

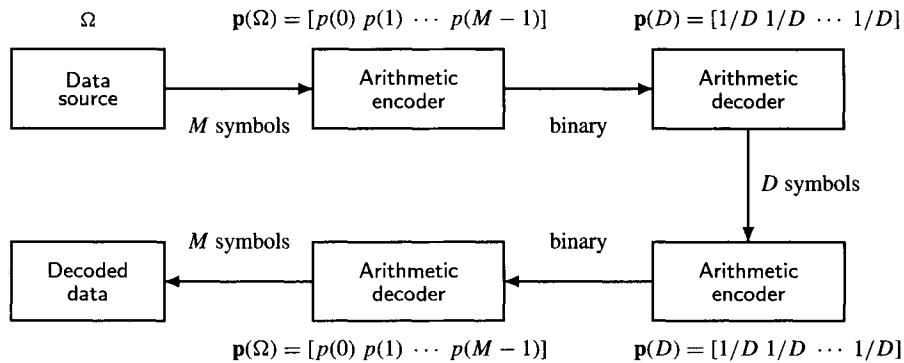
★ Renormalization loop
 ★ Increment symbol counter
 ★ most significant digit
 ★ Update interval base
 ★ Update code value
 ★ Scale interval length

and

$$t(x) = \{n : D^{-n-1} < x \leq D^{-n}\} = \lfloor -\log_D(x) \rfloor. \quad (5.65)$$

Example 5.10. Table 5.5 shows an example of how the arithmetic coding renormalization changes when $D = 16$. Again, the source and data sequence are those of Example 5.3. Comparing these results with those of Example 5.8 (Table 5.3), we can see the substantial reduction in the number of times the intervals are rescaled.

Algorithms 5.9 and 5.10 can also be used for values of D that are not powers of 2, but with inefficient radix- D arithmetic. The problem is that while the multiplications by the interval length can be approximated, rescaling must be exact. For example, if $D = 2$, then multiplication by 2^{-P} is computed exactly with bit-shifts, but exact multiplication by 3^{-P} requires special functions [39].

**FIGURE 5.9**

Configuration for using a standard (binary output) arithmetic encoder to replace an encoder with D -symbol (ternary, decimal, etc.) output.

A better alternative is to use a small trick, shown in Fig. 5.9. We can compress the data sequence using a standard encoder/decoder with binary arithmetic. Next, we “decode” the binary compressed data using a D -symbol alphabet with uniform probability distribution $\mathbf{p} = [1/D \ 1/D \ \dots \ 1/D]$. The uniform distribution does not change the distribution of the code values (and thus will not alter compression), but converts the data to the desired alphabet. Since both processes implemented by the decoder are perfectly reversible, the decoder only has to implement the inverse processes. This takes twice the time for encoding and decoding, but is significantly faster than using radix- D arithmetic.

5.3.1.4 Care with Carries

We have seen in Section 5.3.1 that carry propagation is applied only to the set of outstanding bits, which always starts with a 0-bit and is followed possibly by 1-bits. An example of a set of outstanding bits is

$$\emptyset, \{0\}, \{0, 1\}, \{0, 1, 1\}, \{0, 1, 1, 1\}, \dots, \{0, 1, 1, 1, \dots, 1\}.$$

Clearly, with such simple structure we do not have to save the outstanding bits to know what they are. We can just keep a counter with the number of outstanding bits, which is incremented as new bits are defined during renormalization. We can output (or “flush”) these bits whenever a carry occurs or when a new outstanding 0-bit comes from renormalization.

Note that not all bits put out by rescaling have to be outstanding before becoming settled. For example, if we have $b < b + l \leq 0.5$, we not only know that the next bit is zero, but we know that it cannot possibly be changed by a carry and is thus settled. We can disregard these details when implementing carry propagation in buffers, but not when using counters.

With D -symbol output the set of outstanding symbols starts with a symbol $\sigma \in \{0, 1, \dots, D-2\}$ and is possibly followed by several occurrences of the symbol $D-1$, as shown:

$$\emptyset, \{\sigma\}, \{\sigma, D-1\}, \{\sigma, D-1, D-1\}, \dots, \{\sigma, D-1, D-1, \dots, D-1\}.$$

In this case we can only keep the first symbol σ and a counter with the number of outstanding symbols.

It is important to know that the number of outstanding symbols can grow indefinitely; i.e., we can create distributions and infinitely long data sequences such that bits never become settled. The final code value selection (Algorithm 5.5) settles all outstanding symbols, so that we can periodically restart the encoder to limit the number of outstanding symbols.

There are many choices for dealing with carry propagation. The most common are as follows:

1. Save the outstanding symbols temporarily to a buffer and then implement carry propagation in the buffer. This simple technique is efficient when working with bytes ($D = 256$), but can be inefficient when D is small. It can be used only if we know that the buffer is large enough to fit all the compressed symbols, since all of them can be outstanding.
2. Use a counter to keep track of the outstanding symbols, saving all symbols to a buffer or file as soon as they become settled. There is chance of overflow depending on the number of bits used by the counter (mantissas with 32, 53, and 64 bits allow, respectively, 4×10^9 , 9×10^{15} , and 2×10^{19} outstanding symbols). In practical applications a counter overflow is extremely improbable, especially with adaptive coders.
3. Use carry propagation in a buffer, plus “bit-stuffing” [15, 27] in the following form. Outstanding bits are moved out from the buffer as soon as they become settled. Whenever the number of 1-bits exceeds a threshold (e.g., 16), an artificial zero is added to the compressed bit sequence, forcing the outstanding bits to become settled. The decoder can identify when this happens by comparing the number of consecutive 1-bits read. When it exceeds the threshold, the decoder interprets the next bit not as data, but as carry information. If it is 1, then a carry is propagated in the decoder buffer, and the decoding process continues normally. This technique is used by the Q-coder [27].

5.3.1.5 Alternative Renormalizations

We show in Section 5.2.5.4 that there is a great deal of flexibility in rescaling the intervals, and we present in Section 5.3.1.1 an implementation based on a particular form of renormalization. Other choices lead to renormalized intervals with distinct properties, which had been exploited in several manners by different arithmetic coding implementations. Below we show a few examples.

- We have chosen renormalization (5.64), which produces intervals such that $b \in [0, 1]$, $l \in (1/D, 1]$, and $b + l \in (1/D, 2)$. Its main advantage is that it simplifies carry detection and renormalization, especially when $D > 2$. Note that it has a criterion for when to renormalize that is based only on the interval length.
- The decision of when to renormalize can be based on settled symbols. For example, the method by Rubin [13] keeps intervals such that $b \in [0, 1]$ and $b + l \in (0, 1]$, which are renormalized when the most significant symbol becomes settled, i.e., $\lfloor Db \rfloor = \lfloor D(b + l) \rfloor$, meaning that the outstanding symbols are kept in the registers. To avoid the interval length eventually collapsing to zero, the encoder and decoder rescale and shift the interval when its length gets too small, forcing outstanding symbols to become settled.
- An arithmetic coding implementation that became quite popular was proposed by Witten *et al.* (see [25, 51]). Instead of the base length, it uses the extremes points to represent the interval and keeps it renormalized in such a manner that $b + l \in (0, 1]$. Renormalization occurs whenever bits are settled and also when $0.25 \leq b < b + l \leq 0.75$. Thus, it avoids the precision loss that occurs in [13] and uses counters to keep track of the number of outstanding bits. This technique can be adapted to D -symbol output, but it is not as simple as what we have in Appendix 5.1.

- The Q-coder developed by IBM [27, 28] keeps intervals with $l \in (0.75, 1.5]$. This way we have $l \approx 1$, and we can approximate multiplications with $p \cdot l \approx p$ and $(1 - p) \cdot l \approx l - p$.

5.3.2 Adaptive Coding

Since typical information sources tend to be quite complex, we must have a good model of the data source to achieve optimal compression. There are many techniques for modeling complex sources [14, 25, 29, 43, 45, 46, 49] that decompose the source data in different categories, under the assumption that in each category the source symbols are approximately independent and identically distributed and thus well suited to be compressed with arithmetic coding. In general, we do not know the probabilities of the symbols in each category. Adaptive coding is the estimation of the probabilities of the source symbols during the coding process. In this section we study techniques to efficiently combine arithmetic coding with dynamic probability estimation.

5.3.2.1 Strategies for Computing Symbol Distributions

The most efficient technique for computing distributions depends on the data type. When we are dealing with completely unknown data we may want adaptation to work in a completely automatic manner. In other cases, we can use some knowledge of the data properties to reduce or eliminate the adaptation effort. Below we explain the features of some of the most common strategies for estimating distributions.

- Use a constant distribution that is available before encoding and decoding, normally estimated by gathering statistics in a large number of typical samples. This approach can be used for sources such as English text, or weather data, but it rarely yields the best results because few information sources are so simple as to be modeled by a single distribution. Furthermore, there is very little flexibility (e.g., statistics for English text do not fit Spanish text well). On the other hand, it may work well if the source model is very detailed, and in fact it is the only alternative in some very complex models in which meaningful statistics can only be gathered from a very large amount of data.
- Use predefined distributions with adaptive parameter estimation. For instance, we can assume that the data has Gaussian distribution and estimate only the mean and variance of each symbol. If we allow only a few values for the distribution parameters, then the encoder and decoder can create several vectors with all the distribution values and use them according to their common parameter estimation. See Ref. [49] for an example.
- Use two-pass encoding. A first pass gathers the statistics of the source, and the second pass codes the data with the collected statistics. For decoding, a scaled version of vectors \mathbf{p} or \mathbf{c} must be included at the beginning of the compressed data. For example, a book can be archived (compressed) together with its particular symbol statistics. It is possible to reduce the computational overhead by sharing processes between passes. For example, the first pass can simultaneously gather statistics and convert the data to run-lengths.
- Use a distribution based on the occurrence of symbols previously coded, updating \mathbf{c} with each symbol encoded. We can start with a very approximate distribution (e.g., uniform), and if the probabilities change frequently, we can reset the estimates periodically. This technique, explained in the next section, is quite effective and the most convenient and versatile. However, the constant update of the cumulative distribution can increase the computational complexity considerably. An alternative is to update only the probability vector \mathbf{p} after each encoded symbol and update the cumulative distribution \mathbf{c} less frequently (Section 5.3.2.5).

ALGORITHM 5.11**Procedure Interval_Update** (s, b, l, M, \tilde{C})

1. set $\gamma = l/\tilde{C}(M)$
2. if $s = M - 1$,
 - then set $y \leftarrow b + l$;
 - else set $y \leftarrow b + \gamma \cdot \tilde{C}(s + 1)$;
3. set $\{ b \leftarrow b + \gamma \cdot \tilde{C}(s);$
 $l \leftarrow y - b; \}$
4. return.

- ★ Compute division result
- ★ Special case for last symbol
 - ★ end of interval
 - ★ base of next subinterval
- ★ Update interval base
- ★ Update interval length as difference

5.3.2.2 Direct Update of Cumulative Distributions

After encoding/decoding k symbols the encoder/decoder can estimate the probability of a symbol as

$$p(m) = \frac{\tilde{P}(m)}{k + M}, \quad m = 0, 1, 2, \dots, M - 1, \quad (5.66)$$

where $\tilde{P}(m) > 0$ is the number of times symbol m was encoded/decoded plus 1 (added to avoid zero probabilities). The symbol occurrence counters are initialized with $\tilde{P}(m) = 1$ and incremented after a symbol is encoded. We define the cumulative sum of occurrences as

$$\tilde{C}(m) = \sum_{i=0}^{m-1} \tilde{P}(i), \quad m = 0, 1, 2, \dots, M, \quad (5.67)$$

and the cumulative distribution as

$$c(m) = \frac{\tilde{C}(m)}{k + M} = \frac{\tilde{C}(m)}{\tilde{C}(M)}, \quad m = 0, 1, 2, \dots, M. \quad (5.68)$$

Only a few changes to the algorithms of Section 5.3.1.1 are sufficient to include the ability to dynamically update the cumulative distribution. First, we may use (5.68) to change all multiplications by $c(s)$, as follows:

$$b \leftarrow b + \frac{l \cdot \tilde{C}(s)}{\tilde{C}(M)}. \quad (5.69)$$

Similarly, the changes to the integer-based version in Appendix 5.1 may be in the form:

$$B \leftarrow B + \left\lfloor \frac{L \cdot \tilde{C}(s)}{\tilde{C}(M)} \right\rfloor. \quad (5.70)$$

However, since divisions are much more expensive than additions and multiplications, it is better to compute $\gamma = l/\tilde{C}(M)$ once and implement interval updating as shown in Algorithm 5.11 [50, 51]. The corresponding changes in Algorithm 5.7 are shown in Algorithm 5.12. In Algorithm 5.7 and 5.12 we may need to compute several multiplications, under the assumption they are faster than a single division (see Algorithm 5.21 for a more efficient search method). If this is not true, Algorithm 5.13 shows how to replace those multiplications by one extra division.

After the modifications above, Algorithms 5.1 and 5.6 are made adaptive by adding the following line:

2.4. **Update_Distribution** (s_k, M, \tilde{C});

ALGORITHM 5.12**Function Interval_Selection (v, b, l, M, \tilde{C})**

-
1. set $\{ s \leftarrow M - 1;$
 $\gamma = l/\tilde{C}(M)$
 $x \leftarrow b + \gamma \cdot \tilde{C}(M - 1);$
 $y \leftarrow b + l; \}$
★ Start search from last symbol
★ Compute division result
★ base of search interval
★ end of search interval
 2. while $x > v$ do
 - 2.1. set $\{ s \leftarrow s - 1;$
 $y \leftarrow x;$
 $x \leftarrow b + \gamma \cdot \tilde{C}(s); \}$
★ Sequential search for correct interval
★ Decrement symbol by one
★ Move interval end
★ Compute new interval base
 3. set $\{ b \leftarrow x;$
 $l \leftarrow y - b; \}$
★ Update interval base
★ Update interval length as difference
 4. return $s.$
-

ALGORITHM 5.13**Function Interval_Selection (v, b, l, M, \tilde{C})**

1. set $\{ s \leftarrow M - 1;$
 $\gamma = l/\tilde{C}(M)$
 $W \leftarrow (v - b)/\gamma; \}$
★ Initialize search from last symbol
★ Compute division result
★ Code value scaled by $\tilde{C}(M)$
 2. while $\tilde{C}(s) > W$ do
 - 2.1. set $s \leftarrow s - 1;$
★ Look for correct interval
★ Decrement symbol by one
 3. set $\{ b \leftarrow b + \gamma \cdot \tilde{C}(s);$
 $l \leftarrow \gamma \cdot [\tilde{C}(s + 1) - \tilde{C}(s)]; \}$
★ Update interval base
★ Update interval length
 4. return $s.$
-

ALGORITHM 5.14**Procedure Update_Distribution (s, M, \tilde{C})**

1. for $m = s + 1$ to M do
 - 1.1 set $\tilde{C}(s) \leftarrow \tilde{C}(s) + 1;$
★ For all symbols larger than s
★ Increment cumulative distribution
 2. return.
-

The procedure to update the cumulative distribution is shown in Algorithm 5.14. Note that in this algorithm it is necessary to compute up to M additions. Similarly, in Step 2 of Algorithm 5.13 we must perform up to M comparisons and subtractions. Since the number of operations in both cases decreases with the symbol number, it is good to sort the symbol by increasing probability. Witten *et al.* [25] present an implementation that simultaneously updates the distribution while keeping it sorted.

5.3.2.3 Binary Arithmetic Coding

Binary arithmetic coders work only with a binary source alphabet ($M = 2$). This is an important type of encoder because it helps to solve many of the complexity issues created with the dynamic update of the cumulative distributions (Algorithm 5.14). When $M = 2$ the cumulative distribution

ALGORITHM 5.15**Procedure Binary_Interval_Update** ($s, b, l, \tilde{C}(1), \tilde{C}(2)$)

1. set $x \leftarrow l \cdot \tilde{C}(1)/\tilde{C}(2);$ ★ Point for interval division
 2. if $s = 0,$ ★ If symbol is zero,
 then set $l \leftarrow x;$ ★ Update interval length
 else set $\{ b \leftarrow b + x;$ ★ Move interval base and
 $l \leftarrow l - x; \}$ ★ Update interval length
 3. return.
-

ALGORITHM 5.16**Function Binary_Interval_Selection** ($v, b, l, \tilde{C}(1), \tilde{C}(2)$)

1. set $x \leftarrow l \cdot \tilde{C}(1)/\tilde{C}(2);$ ★ Point for interval division
 2. if $b + x > v,$ ★ Look for correct interval
 then set $\{ s \leftarrow 0;$ ★ Symbol is 0: no change to interval base
 $l \leftarrow x; \};$ ★ Update interval length
 - else set $\{ s \leftarrow 1;$ ★ Symbol is 1:
 $b \leftarrow b + x;$ ★ Move interval base and
 $l \leftarrow l - x; \}$ ★ Update interval length
 3. return $s.$
-

ALGORITHM 5.17**Procedure Update_Binary_Distribution** ($s, \tilde{\mathbf{C}}$)

1. if $s = 0,$ then set $\tilde{C}(1) \leftarrow \tilde{C}(1) + 1;$ ★ If $s = 0,$ then increment 0-symbol counter
 2. set $\tilde{C}(2) \leftarrow \tilde{C}(2) + 1;$ ★ Increment symbol counter
 3. return.
-

vector is simply $\mathbf{c} = [0 \ p(0) \ 1]$, which makes coding and updating the cumulative distribution much simpler tasks. Algorithms 5.15, 5.16, and 5.17, have the procedures for, respectively, binary encoding (interval update), decoding (interval selection and update), and distribution update. Note that instead of using the full vector $\tilde{\mathbf{C}}$ we use only $\tilde{C}(1) = \tilde{P}(0)$ and $\tilde{C}(2) = \tilde{P}(0) + \tilde{P}(1).$ The renormalization procedures do not have to be changed for binary arithmetic coding.

Example 5.11. Binary arithmetic coding has universal application because, just as any number can be represented using bits, data symbols from any alphabet can be coded as a sequence of binary symbols. Figure 5.10 shows how the process of coding data from a 6-symbol source can be decomposed in a series of binary decisions, which can be represented as a *binary search tree* [30]. The leaf nodes correspond to the data source symbols, and intermediate nodes correspond to the decisions shown below them.

Underlined numbers are used for the intermediate nodes, and their value corresponds to the number used for the comparison (they are the “keys” for the binary search tree [30]). For instance, node \underline{m} corresponds to test “ $s < m?$ ” Symbols are coded starting from the root of the tree, and continue until a leaf node is encountered. For example, if we want to code the symbol $s = 2,$ we

start coding the information “ $s < 3?$ ” indicated by node 3; next we go to node 1 and code the information “ $s < 1?$ ” and finally move to node 2 to code “ $s < 2?$ ”. At each node the information is coded with a different set of probabilities, which in Fig. 5.10 are shown below the tree nodes. These probabilities, based on the number of symbol occurrences, are updated with Algorithm 5.17. An alphabet with M symbols needs $M - 1$ probability estimates for the intermediate nodes. The decoder follows the same order, using the same set of probabilities. (Note that this is equivalent to using the scheme explained in Section 5.2.5.3, and shown in Fig. 5.5.)

There is no loss in compression in such scheme. For example, when coding symbol $s = 2$ we can compute the symbol probability as a product of conditional probabilities [22].

$$\begin{aligned}\text{Prob}(s = 2) &= \text{Prob}(s < 3) \cdot \text{Prob}(s = 2 | s < 3) \\ &= \text{Prob}(s < 3) \cdot \text{Prob}(s \geq 1 | s < 3) \cdot \text{Prob}(s = 2 | 1 \leq s < 3) \\ &= \text{Prob}(s < 3) \cdot \text{Prob}(s \geq 1 | s < 3) \cdot \text{Prob}(s \geq 2 | s \geq 1, s < 3).\end{aligned}\quad (5.71)$$

This means that

$$\begin{aligned}\log_2 [\text{Prob}(s = 2)] &= \log_2 [\text{Prob}(s < 3)] + \log_2 [\text{Prob}(s \geq 1 | s < 3)] \\ &\quad + \log_2 [\text{Prob}(s \geq 2 | 1 \leq s < 3)].\end{aligned}\quad (5.72)$$

The left-hand-side of (5.72) is the number of bits required to code symbol $s = 2$ directly with a 6-symbol model, which is equal to the sum of the bits used to code the same symbol by successively coding the binary symbols in the binary-search tree (see Fig. 5.10). We do not have to worry about computing explicit values for the conditional probabilities because, when we use a different adaptive binary model for each node, we automatically get the estimate of the proper conditional probabilities. This property is valid for binary-tree decompositions of any data alphabet.

A binary-search tree as in Fig. 5.10 can be automatically generated using, for example, the bisection algorithm [17, 19, 39]. Algorithms 5.18 and 5.19 show such implementations for encoding, decoding, and overall probability estimation updates. Note that they use the binary coding and decoding functions of Algorithms 5.15 and 5.16, and use only one vector, $\bar{\mathbf{C}}$, with dimension

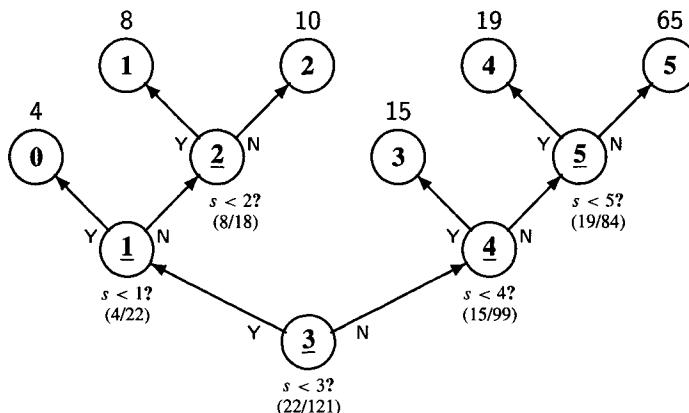


FIGURE 5.10

Example of a binary-search tree with the sequential decisions required for coding data from a 6-symbol alphabet using binary encoders. Leaf nodes represent data symbols, and the numbers above them represent their number of occurrences. The binary information indicated by each question mark is coded with the probability estimate shown in parentheses.

ALGORITHM 5.18**Procedure Interval-Update (s, b, l, M, \bar{C})**

1. set $\{ u \leftarrow 0; n \leftarrow M;$
 $k \leftarrow \bar{C}(0);$
 $\bar{C}(0) \leftarrow \bar{C}(0) + 1; \}$
 2. while $n - u > 1$ do
 - 2.1. set $m \leftarrow \lfloor (u + n)/2 \rfloor;$
 - 2.2. if $s < m$,
 then set $\{ n \leftarrow m;$
 $\text{Binary-Interval-Update}(0, b, l, \bar{C}(m), k);$
 $k \leftarrow \bar{C}(m);$
 $\bar{C}(m) \leftarrow \bar{C}(m) + 1; \}$
 - else set $\{ u \leftarrow m;$
 $\text{Binary-Interval-Update}(1, b, l, \bar{C}(m), k);$
 $k \leftarrow k - \bar{C}(m); \}$
 3. return.
-
- ★ Initialize bisection search limits
 ★ First divisor = symbol counter
 ★ Increment symbol counter
 ★ Bisection search loop
 ★ Compute middle point
 ★ If symbol is smaller than middle,
 ★ then update upper limit
 ★ Code symbol 0
 ★ Set next divisor
 ★ Increment 0-symbol counter
 ★ else update lower limit
 ★ Code symbol 1
 ★ Set next divisor

ALGORITHM 5.19**Function Interval-Selection (v, b, l, M, \bar{C})**

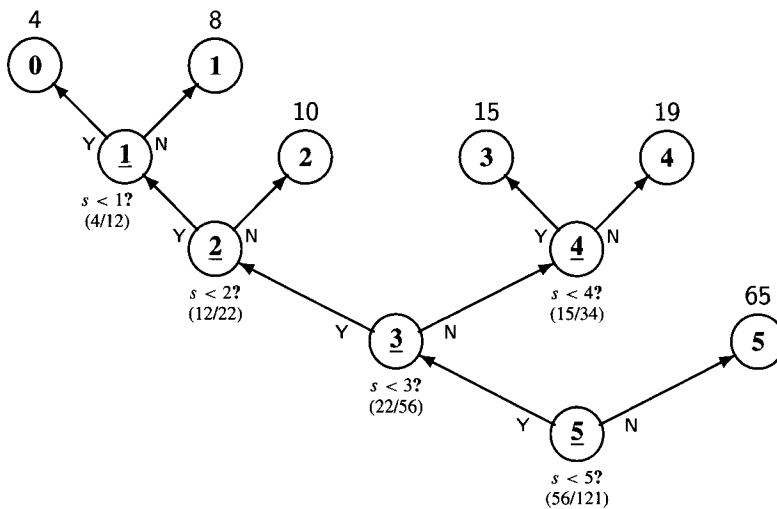
1. set $\{ s \leftarrow 0; n \leftarrow M;$
 $k \leftarrow \bar{C}(0);$
 $\bar{C}(0) \leftarrow \bar{C}(0) + 1; \}$
 2. while $n - s > 1$ do
 - 2.1. set $m \leftarrow \lfloor (s + n)/2 \rfloor;$
 - 2.2. if $\text{Binary-Interval-Selection}(v, b, l, \bar{C}(m), k) = 0$,
 then set $\{ n \leftarrow m;$
 $k \leftarrow \bar{C}(m);$
 $\bar{C}(m) \leftarrow \bar{C}(m) + 1; \}$
 - else set $\{ s \leftarrow m;$
 $k \leftarrow k - \bar{C}(m); \}$
 3. return s .
-
- ★ Initialize bisection search bounds
 ★ First divisor = symbol counter
 ★ Increment symbol counter
 ★ Bisection search loop
 ★ Compute middle point
 ★ If symbol is smaller than
 ★ middle,
 ★ then update upper limit
 ★ Set next divisor
 ★ Increment 0-symbol counter
 ★ else update lower limit
 ★ Set next divisor

$M - 1$, to store the tree-branch occurrence counters. Each component $\bar{C}(m)$, $0 < m < M$, contains the number of the number of times we had “ $s < m?$ ” on tree node \underline{m} , and $\bar{C}(0)$ contains the total number of symbols coded. This vector is initialized with the number of leaf nodes reachable from the left branch of the corresponding node.

The binary conditional probabilities estimates are computed during the encoding and decoding directly from \bar{C} . For instance, in the example of Fig. 5.10 we have

$$\bar{C} = [121 \ 4 \ 8 \ 22 \ 15 \ 19].$$

In order to code the decision at node 3 we use probability estimate $22/121$, which is defined by counters $\bar{C}(0) = 121$ and $\bar{C}(3) = 22$. If we move to node 1, then the next probability estimate is $\bar{C}(1)/\bar{C}(3) = 4/22$. On the other hand, if we move node 4, we need to use probability estimate $\bar{C}(4)/[\bar{C}(0) - \bar{C}(3)] = 15/99$, which is also readily computed from components of \bar{C} . See Algorithms 5.18 and 5.19 for details.

**FIGURE 5.11**

Another example of a binary-search tree with for coding data from a 6-symbol source. The number of symbol occurrences is the same as shown in Fig. 5.10. This tree has been designed to minimize the average number of binary symbols coded.

Since we use bisection search in Algorithms 5.18 and 5.19, the number of times the binary encoding and decoding functions are called is between $\lfloor \log_2 M \rfloor$ and $\lceil \log_2 M \rceil$, for all data symbols. Thus, by using binary-tree searches and binary arithmetic coding we greatly reduce the *worst-case* complexity required to update probability estimates.

Example 5.12. Figure 5.11 shows a second example of a binary-search tree that can be used to code the 6-symbol data of Example 5.10. Different search algorithms can be used to create different trees. For example, we could have used a sequential search, composed of tests, “ $s < 5?$ ”, “ $s < 4?$ ”, “ $s < 3?$ ”, . . . , “ $s < 1?$ ”

The trees created from bisection search minimize the maximum number of binary symbols to be coded, but not the average. The tree of Fig. 5.11 was designed so that the most frequent symbols are reached with the smallest number of tests. Table 5.6 shows the average number of symbols required for coding symbols from the source of Example 5.11 (considering probability estimates as the actual probabilities), for different trees created from different search methods.

Because we have the symbols sorted by increasing probability, the performance of the tree defined by sequential search, starting from the most probable symbol, is quite good. The tree of Fig. 5.11 is the one that minimizes the average number of coded binary symbols. Below we explain how it is designed.

Prefix coding [4, 5, 21, 32, 55, 56] is the process of coding information using decision trees as defined above. The coding process we have shown above is identical, except that we call a binary arithmetic encoding/decoding function at each node. Thus, we can use all the known facts about prefix coding to analyze the computational complexity of binary arithmetic encoding/decoding, if we measure complexity by the number of coded binary symbols.

Since the optimal trees for prefix coding are created using the Huffman algorithm [2], these trees are also optimal for binary arithmetic encoding/decoding [23]. Strictly speaking, if the data symbols are not sorted according to their probability, the optimal Huffman tree does not satisfy the requirements for binary-search trees, i.e., “keys” are not properly sorted, and we cannot define

Table 5.6 Number of Binary Symbols Coded Using Trees Created from Different Types of Binary Searches, Applied to Data Source of Example 5.10

Data Symbol <i>s</i>	Probability Estimate <i>p(s)</i>	Number of Binary Symbols Coded					
		Sequential Search		Bisection Search		Optimal Search	
		<i>N(s)</i>	<i>p(s)N(s)</i>	<i>N(s)</i>	<i>p(s)N(s)</i>	<i>N(s)</i>	<i>p(s)N(s)</i>
0	0.033	6	0.198	2	0.066	4	0.132
1	0.066	5	0.331	3	0.198	4	0.264
2	0.083	4	0.331	3	0.248	3	0.248
3	0.124	3	0.372	2	0.248	3	0.372
4	0.157	2	0.314	3	0.471	3	0.471
5	0.537	1	0.537	3	1.612	1	0.537
Sum	1.000	21	2.083	16	2.843	18	2.025

Note: The trees corresponding to bisection and optimal searches are shown in Figs. 5.10 and 5.11, respectively.

a node with a simple comparison of the type “*s* < *m*?” This problem is solved by storing the paths from the root node to leaf nodes, i.e., the Huffman codewords.

5.3.2.4 Tree-Based Update of Cumulative Distributions

In this section, we show that we can use binary-search trees (Section 5.3.2.3) to efficiently combine computing the cumulative distribution, updating it, encoding, and decoding, without having to use a binary arithmetic encoder. We present techniques similar to the methods proposed by Moffat [31] and Fenwick [42]. We start with an example of how to compute the cumulative distribution vector $\tilde{\mathbf{C}}$ from the statistics $\tilde{\mathbf{C}}$ gathered while using the binary search trees.

Example 5.13. Let us consider the binary search tree shown in Fig. 5.10. Let us assume that we had been using Algorithms 5.18 and 5.19 to compute the number of symbols occurrences in the tree, $\tilde{\mathbf{C}}$, and we want to compute the cumulative distribution $\tilde{\mathbf{C}}$ from $\tilde{\mathbf{C}} = [121 \ 4 \ 8 \ 22 \ 15 \ 19]$.

From the tree structure we can find out that, except for the root node, the counter at each node has the number of occurrences of all symbols found following the left branch, i.e.,

$$\begin{aligned}\tilde{C}(0) &\equiv \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) = 121 \\ \tilde{C}(1) &= \tilde{P}(0) = 4 \\ \tilde{C}(2) &= \tilde{P}(1) = 8 \\ \tilde{C}(3) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) = 22 \\ \tilde{C}(4) &= \tilde{P}(3) = 15 \\ \tilde{C}(5) &= \tilde{P}(4) = 19,\end{aligned}$$

where $\tilde{P}(s)$ is the number of time symbol *s* has occurred. From these equations we can compute the vector with cumulative distributions as

$$\begin{aligned}\tilde{C}(0) &\equiv 0 \\ \tilde{C}(1) &= \tilde{P}(0) = \tilde{C}(1) = 4 \\ \tilde{C}(2) &= \tilde{P}(0) + \tilde{P}(1) = \tilde{C}(1) + \tilde{C}(2) = 12 \\ \tilde{C}(3) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) = \tilde{C}(3) = 22 \\ \tilde{C}(4) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) = \tilde{C}(3) + \tilde{C}(4) = 37 \\ \tilde{C}(5) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) = \tilde{C}(3) + \tilde{C}(4) + \tilde{C}(5) = 56 \\ \tilde{C}(6) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) = \tilde{C}(0) = 121.\end{aligned}$$

We can do the same with the tree of Fig. 5.11 and find different sets of equations. In this case the counters are

$$\begin{aligned}
 \tilde{C}(0) &\equiv \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) &= 121 \\
 \tilde{C}(1) &= \tilde{P}(0) &= 4 \\
 \tilde{C}(2) &= \tilde{P}(0) + \tilde{P}(1) &= 12 \\
 \tilde{C}(3) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) &= 22 \\
 \tilde{C}(4) &= \tilde{P}(3) &= 15 \\
 \tilde{C}(5) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) &= 56
 \end{aligned}$$

and their relationship with the cumulative distribution is given by

$$\begin{aligned}
 \tilde{C}(0) &\equiv 0 \\
 \tilde{C}(1) &= \tilde{P}(0) &= \tilde{C}(1) &= 4 \\
 \tilde{C}(2) &= \tilde{P}(0) + \tilde{P}(1) &= \tilde{C}(2) &= 12 \\
 \tilde{C}(3) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) &= \tilde{C}(3) &= 22 \\
 \tilde{C}(4) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) &= \tilde{C}(3) + \tilde{C}(4) &= 37 \\
 \tilde{C}(5) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) &= \tilde{C}(5) &= 56 \\
 \tilde{C}(6) &= \tilde{P}(0) + \tilde{P}(1) + \tilde{P}(2) + \tilde{P}(3) + \tilde{P}(4) + \tilde{P}(5) &= \tilde{C}(0) &= 121.
 \end{aligned}$$

The equations that we obtain from any decision tree are linearly independent, and thus it is always possible to compute the cumulative distribution $\tilde{\mathbf{C}}$ from the counters $\tilde{\mathbf{C}}$. We show next how to use the tree structure to efficiently compute the components of $\tilde{\mathbf{C}}$ required for encoding symbol s , $\tilde{C}(s)$, and $\tilde{C}(s+1)$.

In order to compute $\tilde{C}(s)$, when we move from the root of the tree, up to the leaf representing symbol s , we simply add the value of $\tilde{C}(n)$ for each node n that does not have its condition satisfied (i.e., we move up its right-side branch). For example, to compute $\tilde{C}(2)$ using the tree of Fig. 5.10, we start from node 3, and move left to node 1, right to node 2, and right to leaf 2. Since we move along the right branch of nodes 1 and 2, we conclude that $\tilde{C}(2) = \tilde{C}(1) + \tilde{C}(2) = 12$.

The value of $\tilde{C}(s+1)$ can be computed together with $\tilde{C}(s)$: we just have to add the running sum for $\tilde{C}(s)$ and $\tilde{C}(n)$ at the last left-side branch taken before reaching leaf s . For example, when computing $\tilde{C}(2)$, the last left-side branch is taken at root node 3, and thus $\tilde{C}(3)$ is equal to the running sum (zero) plus $\tilde{C}(3) = 22$.

Algorithms 5.20 and 5.21 show how to combine all the techniques above to simultaneously compute and update the cumulative distribution and use the computed values for encoding and decoding. They use a tree defined by bisection search, but it is easy to modify them to other tree structures. After Step 2 of Algorithm 5.20 we have $E = \tilde{C}(s)$ and $F = \tilde{C}(s+1)$ computed and updated with a number of additions proportional to $\log_2(M)$.

5.3.2.5 Periodic Updates of the Cumulative Distribution

We have seen in Section 5.3.2.2 that adaptive coding can increase the arithmetic coding computational complexity significantly, because of the effort to update the cumulative distributions. In Sections 5.3.2.3 and 5.3.2.4 we present tree-based updating techniques that reduce this complexity very significantly. However, adaptation can still be a very significant fraction of the overall coding computational complexity, and it happens that there is not much we can do if we insist on the assumption that the probability model has to be updated immediately after each encoded/decoded symbol; i.e., estimates are in the form given by (5.68), with a division by the factor $\tilde{C}(M)$.

However, with accurate and fast source modeling we can avoid having probabilities changing so quickly that we need to refresh estimates on a symbol-by-symbol basis. For example, an image

ALGORITHM 5.20**Procedure Interval_Update** (s, b, l, M, \bar{C})

1. set $\{ u \leftarrow 0; n \leftarrow M;$
 $E \leftarrow 0; F \leftarrow \bar{C}(0);$
 $\gamma \leftarrow l/\bar{C}(0); \}$
 2. while $n - u > 1$ do
 - 2.1. set $m \leftarrow \lfloor (u + n)/2 \rfloor;$
 - 2.2. if $s < m$,
 then set $\{ n \leftarrow m;$
 $F \leftarrow E + \bar{C}(m);$
 $\bar{C}(m) \leftarrow \bar{C}(m) + 1; \}$
 - else set $\{ u \leftarrow m;$
 $E \leftarrow E + \bar{C}(m); \}$
 3. if $s = M - 1$,
then set $y \leftarrow b + l;$
else set $y \leftarrow b + \gamma \cdot F;$
 4. set $\{ b \leftarrow b + \gamma \cdot E;$
 $l \leftarrow y - b;$
 $\bar{C}(0) \leftarrow \bar{C}(0) + 1; \}$
 5. return.
- ★ Initialize bisection search limits
★ Initialize cumulative sum bounds
★ Compute result of division
★ Bisection search loop
★ Compute middle point
★ If symbol is smaller than middle,
 ★ then update bisection upper limit
★ Set upper bound on cumulative sum
★ Increment node occurrence counter
★ else update bisection lower limit
★ Increment lower bound on cum. sum
★ Set interval end according to symbol
★ exact multiplication
★ base of next subinterval
★ Update interval base
★ Update interval length as difference
★ Increment symbol counter

ALGORITHM 5.21**Function Interval_Selection** (v, b, l, M, \bar{C})

1. set $\{ s \leftarrow 0; n \leftarrow M;$
 $x \leftarrow b; y \leftarrow b + l;$
 $E \leftarrow 0;$
 $\gamma \leftarrow l/\bar{C}(0); \}$
 2. while $n - s > 1$ do
 - 2.1. set $\{ m \leftarrow \lfloor (s + n)/2 \rfloor;$
 $z \leftarrow b + \gamma \cdot [E + \bar{C}(m)]; \}$
 - 2.2. if $z > v$,
then set $\{ n \leftarrow m;$
 $y \leftarrow z;$
 $\bar{C}(m) \leftarrow \bar{C}(m) + 1; \}$
 - else set $\{ s \leftarrow m;$
 $x \leftarrow z;$
 $E \leftarrow E + \bar{C}(m); \}$
 3. set $\{ b \leftarrow x;$
 $l \leftarrow y - x;$
 $\bar{C}(0) \leftarrow \bar{C}(0) + 1; \}$
 4. return $s.$
- ★ Initialize bisection search symbol limits
★ Set search interval bounds
★ Initialize cumulative sum
★ Compute result of division
★ Bisection search loop
★ Compute middle point
★ Value at middle point
★ If symbol is smaller than middle,
 ★ then update bisection upper limit
★ new interval end
★ Increment node occurrence counter
★ else update bisection lower limit
★ new interval base
★ increment lower bound on cumulative sum
★ Update interval base
★ Update interval length as difference
★ Increment symbol counter

encoder may use different probabilities depending on a classification of the part being encoded, which can be something like “constant,” “smooth,” “edge,” “high-frequency pattern,” etc. With these techniques, we can assume that the source state may change quickly, but the source properties (symbol probabilities) for each state change slowly.

Under these assumptions, and unless the number of data symbols is very large (thousands or millions [51]), a significantly more efficient form of adaptive arithmetic coding updates only the vector with symbol occurrence counters ($\tilde{\mathbf{P}}$) after each symbol and updates the distribution estimate (\mathbf{c}) periodically or following some special events. For example, the Q-coder updates its probability estimation only during renormalization [27]. For periodic updates, we define R as the number of symbols coded between updates of \mathbf{c} . Typically, the period R is a small multiple of M (e.g., $R = 4M$), but to improve the accuracy while minimizing the computations, we can start with frequent updates and then decrease their frequency.

One immediate consequence of this approach is that while coding we can use the simpler procedures of Section 5.3.1.1 and Appendix 5.1 and can completely avoid the divisions in Eqs. (5.68) to (5.70). In addition, if the period R is large enough, then it is feasible to do many complex tasks while updating \mathbf{c} , in order to increase the encoding/decoding speed. For instance, we can sort the symbols according to their probability or find the optimal (Huffman) tree for decoding.

5.3.3 Complexity Analysis

Large computational complexity had always been a barrier to the adoption of arithmetic coding. In fact, for many years after arithmetic coding was invented, it was considered little more than a mathematical curiosity because the additions made it slow, multiplications made it very expensive, and divisions made it impractical. In this section, we analyze the complexity of arithmetic coding and explain how the technological advances that give us fast arithmetic operations change the complexity analysis.

We have to observe that the *relative* computational complexity of different coding operations changed dramatically. Arithmetic operations today are much more efficient and not only due to the great increase in the CPU clock frequency. In the past, the ratio between clock cycles used by some simple operations (comparisons, bit-shifts, table look-up) and arithmetic operations (specially division) was quite large. Today, this ratio is much smaller [58–61], invalidating previous assumptions for complexity reduction. For instance, a set of comparisons, bit-shifts, and table look-up now takes significantly more time than a multiplication.

In this section we use the “big- O ” notation of [30], where $O(\cdot)$ indicates asymptotic upper bounds on the computational complexity.

The main factors that influence complexity are as follows:

- Interval renormalization and compressed data input and output.
- Symbol search.
- Statistics estimation (adaptive models only).
- Arithmetic operations.

In the next sections we analyze each of these in some detail. However, we will not consider special hardware implementations (ASICs) for three reasons: (a) it is definitely outside the scope of this text; (b) our model also applies to some specialized hardware, like DSP chips; (c) many optimization techniques for general CPUs also apply to efficient hardware.

5.3.3.1 Interval Renormalization and Compressed Data Input and Output

In integer-based implementations the interval renormalization can be calculated with only bit-shifts. However, when $D = 2$, renormalization occurs quite frequently, consuming many clock cycles. Using larger output alphabets in the form $D = 2^r$ reduces the frequency of the renormalization by a factor of r (see Example 5.10) and thus may reduce the number of clock cycles used

for renormalization very significantly. Floating-point implementations may need multiplication during renormalization, but if D is large enough, then these operations do not add much to the overall complexity.

Data input and output, which occur during renormalization, also have significant impact on the encoding and decoding speed. Since computers and communication systems work with groups of 8 bits (bytes), processing 1 bit at a time requires extra clock cycles to properly align the data. Thus, there is substantial speed-up when renormalization produces bits that can be easily aligned in bytes (e.g., $D = 2^4$, $D = 2^8$, or $D = 2^{16}$). The case $D = 256$ has been shown to be particularly efficient [52, 54].

5.3.3.2 Symbol Search

The computational complexity of the arithmetic decoder may be many times larger than the encoder's because the decoder needs to solve (5.16), i.e., find out in which subinterval the current code value is. This is a line-search problem [17, 30], where we need to minimize both the number of points tested and the computations used for determining these points. The possible difference in complexity between the encoder and decoder grows with the number of data symbols, M . We can see from Algorithms 5.15 and 5.16 that when $M = 2$, the complexity is practically the same for the encoder and decoder.

There are four main schemes for symbol search that are described below.

5.3.3.2.1 Sequential Search on Sorted Symbols This search method, used in Algorithm 5.7 and Algorithm 5.12, in the worst-case searches $M - 1$ intervals to find the decoded symbol. We can try to improve the average performance by sorting the symbols according to their probability. Assuming that the symbols are sorted with increasing probability, the average number of tests is

$$\bar{N}_s(\Omega) = \sum_{m=0}^{M-1} p(m)(M-m) = M - \sum_{m=0}^{M-1} mp(m) = M - \bar{s}(\Omega) \leq M, \quad (5.73)$$

where $\bar{s}(\Omega)$ is the average symbol number (after sorting) put out by data source Ω . Thus, if the symbol distribution is very skewed ($\bar{s}(\Omega) \approx M - 1$), then sequential search can be efficient.

5.3.3.2.2 Bisection Search With this type of search, shown in Algorithms 5.19, 5.21, and 5.28, the number of tests is bounded by

$$\lfloor \log_2(M) \rfloor \leq \bar{N}_b(\Omega) \leq \lceil \log_2(M) \rceil, \quad (5.74)$$

independently of the probabilities of the data symbols. Note that the *encoder* may also have to implement the bisection search when it is used with binary coding (Section 5.3.2.3) or when using trees for updating the probability estimates (Section 5.3.2.4).

5.3.3.2.3 Optimal Tree-Based Search We show in Section 5.3.2.4 how the process of encoding and decoding data from an M -symbol alphabet can be decomposed in a set of binary decisions, and in Example 5.12 we show its similarities to prefix coding and an optimal decision tree designed with the Huffman algorithm. The interval search process during decoding is also defined by binary decisions, and the optimal set of decisions is also computed with the Huffman algorithm [2, 23]. Thus, we can conclude that the average number of tests in such scheme is bounded as the average number of bits used to code the source with Huffman coding, which is

given by [9]:

$$\max\{1, H(\Omega)\} \leq \bar{N}_o(\Omega) \leq H(\Omega) + 0.086 + \max_{m=0,1,\dots,M-1}\{p(m)\}. \quad (5.75)$$

Implementing the optimal binary-search tree requires some extra storage, corresponding to the data normally used for representing a Huffman code, and it is necessary to reorder the symbols according to probability (as in the example of Fig. 5.11) or use a modified definition of the cumulative distribution.

With adaptive coding we have the problem that the optimal tree is defined from the symbol probabilities, which are unknown when encoding and decoding start. This problem can be solved by periodically redesigning the tree, together with the distribution updates (Section 5.3.2.5).

5.3.3.2.4 Bisection Search on Sorted Symbols We can combine the simplicity of bisection with a technique that takes into account the symbol probabilities. When the data symbols are sorted with increasing probability, we can look for the symbol m such that $c(m) \approx 0.5$ and use it as the first symbol (instead of middle point) to divide the interval during bisection. If the distribution is nearly uniform, then this symbol should be near the middle and the performance is similar to standard bisection search. On the other hand, if the distribution is highly skewed, then $m \approx M - 1$, meaning that the most probable symbols are tested first, reducing the average number of tests.

We can extend this technique to find also find the symbols with $c(m)$ near 0.25 and 0.75, and then 0.125, 0.375, 0.625, and 0.825, and so on. Figure 5.12 shows an example of a cumulative distribution, and the process of finding the symbols for initializing the bisection search. The corresponding first levels of the binary-search tree are shown in Fig. 5.13. The decoder does not have to use all levels of that tree; it can use only the first level (and store one symbol), or only the first two levels (and store three symbols), or any desired number of levels. In Fig. 5.13, the complete binary-search tree is defined by applying standard bisection search following the tests shown in the figure. If the encoder uses up to $V > 1$ levels of the binary-search tree defined by

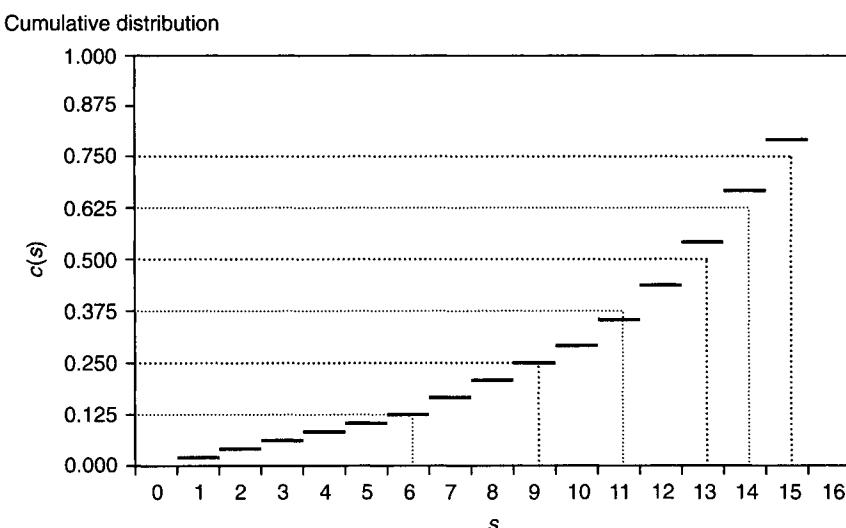
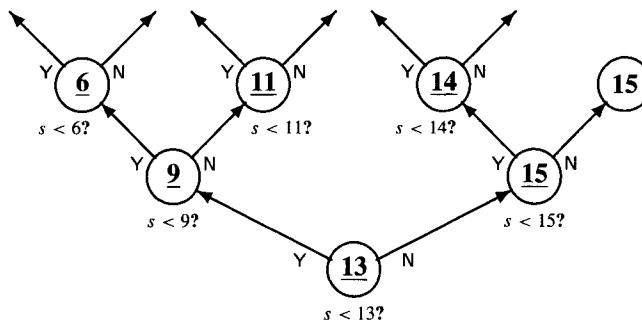


FIGURE 5.12

Technique to find quasi-optimal points to initialize bisection search on sorted symbols.

**FIGURE 5.13**

Decision process corresponding to the quasi-optimal binary-search initialization of Fig. 5.12.

the cumulative distribution, the average number of tests is near the optimal (5.75), and the worst case is not much larger than (5.74).

5.3.3.3 Cumulative Distribution Estimation

Adaptive arithmetic coding requires an extra effort to update the cumulative distribution estimates after coding each symbol, or in a periodic manner. Section 5.3 covers all the important aspects of how adaptive encoders efficiently update the cumulative distributions \mathbf{c} . We analyze two main strategies, depending on whether the cumulative distribution updating is done together with or independently of the symbol search (Section 5.3.3.2).

5.3.3.3.1 Combined Updating, Coding, and Decoding In this case, the most efficient implementations use binary-search trees, and updating is applied to vector $\tilde{\mathbf{C}}$, instead of $\tilde{\mathbf{C}}$. Binary coding (Section 5.3.2.3) and tree-based updating (Section 5.3.2.4) have the same asymptotic complexity, depending on the tree being used. In addition, by comparing Algorithms 5.18 and 5.19 and Algorithms 5.20 and 5.21, we can see that the encoder and the decoder have very similar computational complexity.

The worst-case effort is minimized when we use bisection search, resulting in a number of operations per coded symbol proportional to $\log_2(M)$ (Eq. (5.74)). Huffman trees, which optimize the average performance, require an average number of operations proportional to $\bar{N}_o(\Omega)$, defined in Eq. (5.75).

5.3.3.3.2 Independent Updating When source Ω has low entropy, some symbols should occur much more frequently, and it may be efficient to update $\tilde{\mathbf{C}}$ directly, if the symbols are sorted by increasing probabilities. Implementations by Witten *et al.* [25, 51] use this strategy. However, when all symbols are equally probable, the effort for sorting and updating $\tilde{\mathbf{C}}$ is on average proportional to $M/2$ and in the worst-case proportional to $M - 1$.

As explained in Section 5.3.2.5, with periodic updates of $\tilde{\mathbf{C}}$ we can recompute and sort symbols with reasonable worst-case complexity. We assume that the updating period is R coded symbols. One choice is keep the data symbols not sorted and to update $\tilde{\mathbf{C}}$ using (5.68), which requires $O(M)$ operations per update and an average complexity of $O(M/R)$ operations per coded symbol. Decoding can use bisection for symbol search. If we choose, for example, $R = M$, then we have $O(1)$ operations per update, which is quite reasonable.

Sorting the symbols according to their probability during each update can be done with $O(M \log(M)/R)$ operations per symbol, in the worst case [30]. Since symbols are normally

Table 5.7 Computational Complexity of Symbol Search and Adaptation (Cumulative Distribution Updating) for Fixed and Different Adaptive Arithmetic Coding Techniques

Searching and Updating Methods	Symbol Encoding	Decoder Interval Search	Distribution Updating	Divisions
Fixed code optimal search	$O(1)$	$O(H(\Omega) + 1)$	NA	NA
Sequential search on sorted symbols	$O(1)$	$O(M)$	$O(M)$	$O(1)$
Combined updating and coding, bisection tree	$O(\log(M))$	$O(\log(M))$	$O(\log(M))$	$O(1)$
Combined updating and coding, optimal tree	$O(H(\Omega) + 1)$	$O(H(\Omega) + 1)$	$O(H(\Omega) + 1)$	$O(1)$
Periodic update, bisection search	$O(1)$	$O(\log(M))$	$O(M/R)$	$O(1/R)$
Periodic update, optimal search	$O(1)$	$O(H(\Omega) + 1)$	$O(M \log(M)/R)$	$O(1/R)$

Note: Typically, $R = 4M$ minimizes the complexity without degrading performance.

already sorted from previous updating passes, insertion sort [30] typically can be done with an average of $O(M/R)$ operations. When the symbols are sorted, we can use the more efficient symbol search of Section 5.3.3.2.4. Choosing $R = M$ results in $O(\log(M))$ operations per symbol.

With periodic updating we can both sort symbols, and find the optimal search strategy (Huffman tree), with a reasonable complexity of $O(M \log(M)/R)$ operations per symbol [30]. However, even though the asymptotic complexity of finding the Huffman code is the same as simply sorting, it requires considerable more effort.

Table 5.7 presents a summary of the results above. Note that for the complexity analysis of optimal searches, we can use $O(H(\Omega) + 1)$, instead of the more complex and tighter bounds given by (5.75). The last columns in Table 5.7 indicates the need to use divisions in the form (5.68). Changing the cumulative distribution every symbol requires using different divisors for each coded symbol. Periodic updating, on the other hand, allows us to compute the inverse of divisor once and use it for coding several (R) symbols.

5.3.3.4 Arithmetic Operations

Currently, additions and multiplications are not much slower than other operations, such as bit-shifts and comparison. Divisions, on the other hand, have a much longer latency (number of clock cycles required to perform the operation) and cannot be streamlined with other operations (like special circuitry for multiply-add) [58–61].

First, let us consider the arithmetic required by fixed coders and adaptive coders with periodic updating. The arithmetic coding recursion in the forms (5.9) and (5.40) require two multiplications and, respectively, one and three additions. Thus, except for processor with very slow multiplication, encoding requirements are quite reasonable. Decoding is more complex due to the effort to find the correct decoding interval. If we use (5.16) for interval search, we have one extra division, plus several comparisons (Section 5.3.3.2). The multiplication-only version (5.38) requires several multiplications and comparisons, but with reasonably efficient symbol search this is faster than (5.16) and eliminates the need to define division when multiplications are approximated.

Adaptive coding can be considerably slower because of the divisions in (5.69) and (5.70). They can add up to two divisions per interval updating. In Algorithms 5.11, 5.12, and 5.13, we show how to avoid one of the divisions. Furthermore, updating the cumulative distribution may require a significant number of additions.

Note that having only periodic updates of the cumulative distribution significantly reduces this adaptation overhead, because all these divisions and additions are not required for every coded symbol. For example, a single division, for the computation of $1/\tilde{C}(M)$, plus a number of multiplications and additions proportional to M , may be needed per update. If the update occurs every M coded symbols, then the average number of divisions per coded symbol is proportional to $1/M$, and the number of multiplications and additions are constants.

5.3.4 Further Reading

We presented the coding method that is now commonly associated with the name arithmetic coding, but the reader should be aware that other types of arithmetic coding had been proposed [6, 7]. Standard implementations of arithmetic coding had been defined in some international standards [33, 36, 38, 44]. Several techniques for arithmetic coding complexity reduction not covered here are in the references [11, 18, 23, 25, 27, 28, 34, 37, 42, 51, 52]. We did not mention the fact that errors in arithmetic coded streams commonly lead to catastrophic error propagation, and thus error-resilient arithmetic coding [57] is very important in some applications.

APPENDIX 5.1

Integer-Based Implementation

The following algorithms show the required adaptations in the algorithms in Section 5.3.1.1 for use with integer arithmetic, and with a D -symbol output alphabet. Typically D is a small power of 2, such as 2, 4, 16, or 256. As explained in Section 5.3.1.2, we assume that all numbers are represented as integers, but here we define $B = D^P b$, $L = D^P l$, and $C(s) = D^P c(s)$. In addition, we assume that multiplications are computed with $2P$ digits and results are truncated to P digits. Renormalization (5.64) sets $L > D^{P-1}$, and thus the minimum probability allowed is defined by

$$\lfloor [C(s+1) - C(s)] D^{-P} L \rfloor \geq 1 \Rightarrow C(s+1) - C(s) \geq D; \quad (5.76)$$

i.e., $p(s) \geq D^{1-P}$.

Algorithm 5.22, for integer arithmetic, is almost identical to Algorithm 5.1, except for the initialization of L and the decision for renormalization. The arithmetic operations that update the

ALGORITHM 5.22

Function Arithmetic_Encoder ($N, S, M, \mathbf{C}, \mathbf{d}$)

1. set $\{ B \leftarrow 0; L \leftarrow D^P - 1;$ ★ Initialize interval
 $t \leftarrow 0; \}$ ★ and symbol counter
 2. for $k = 1$ to N do
 - 2.1. Interval_Update (s_k, M, B, L, \mathbf{C}); ★ Encode N data symbols
 - 2.2. if $L < D^{P-1}$, then ★ Update interval according to symbol
 $\text{Encoder_Renormalization}(B, L, t, \mathbf{d});$ ★ If interval is small enough,
★ then renormalize interval
 3. Code_Value_Selection (B, L, t, \mathbf{d}); ★ Choose final code value
 4. return $t.$ ★ Return number of code symbols
-

ALGORITHM 5.23**Procedure Interval_Update (s, M, B, L, C)**

-
1. if $s = M - 1$,
then set $Y \leftarrow L$;
else set $Y \leftarrow \lfloor [L \cdot C(s + 1)] \cdot D^{-P} \rfloor$;
 2. set { $A \leftarrow B$;
 $X \leftarrow \lfloor [L \cdot C(s)] \cdot D^{-P} \rfloor$;
 $B \leftarrow (B + X) \bmod D^P$;
 $L \leftarrow Y - X$; }
 3. if $A > B$, then Propagate_Carry (t, d);
 4. return.
- * If s is last symbol, set first product
 * equal to current interval length
 * or else equal to computed value
 * Save current base
 * Compute second product
 * Update interval base
 * Update interval length
 * Propagate carry if overflow
-

ALGORITHM 5.24**Procedure Encoder_Renormalization (B, L, t, d)**

1. while $L < D^{P-1}$ do
 - 1.1. set { $t \leftarrow t + 1$;
 $d(t) \leftarrow \lfloor B \cdot D^{1-P} \rfloor$;
 $L \leftarrow (D \cdot L) \bmod D^P$;
 $B \leftarrow (D \cdot B) \bmod D^P$; }
 2. return.
- * Renormalization loop
 * Increment symbol counter
 * Output symbol
 * Update interval length
 * Update interval base
-

ALGORITHM 5.25**Procedure Propagate_Carry (t, d)**

1. set $n \leftarrow t$;
 2. while $d(n) = D - 1$ do
 - 2.1. set { $d(n) \leftarrow 0$;
 $n \leftarrow n - 1$; }
 3. set $d(n) \leftarrow d(n) + 1$;
 4. return.
- * Initialize pointer to last outstanding symbol
 * while carry propagation
 * Set outstanding symbol to zero
 * Move to previous symbol
 * Increment first outstanding symbol
-

interval base may overflow the integer registers. To make clear that this is acceptable, we define the results modulo D^P , as in Algorithm 5.23. The results of the multiplications are multiplied by D^{-P} and truncated, meaning that the least significant bits are discarded. Overflow is here detected by a reduction in the base value. For that reason, we implement carry propagation together with the interval update in Algorithm 5.23.

Note that in the renormalization of Algorithm 5.24, we assume that D is a power of 2, and all multiplications and divisions are actually implemented with bit-shifts. The carry propagation with a D -symbol output, shown in Algorithm 5.25, is very similar to Algorithm 5.3.

In Algorithm 5.5 the code value selection is made to minimize the number of bits, assuming that the decoder pads the buffer d with sufficient zeros. This inconvenience can be avoided by simply adding a proper extra symbol at the end of the compressed data. Algorithm 5.26 shows the required modifications. It shifts the interval base by a small amount and resets the interval

ALGORITHM 5.26**Procedure Code_Value_Selection (B, L, t, \mathbf{d})**

1. set { $A \leftarrow B$; ★ Save current base
 $B \leftarrow (B + D^{P-1}/2) \bmod D^P$; ★ Increase interval base
 $L \leftarrow D^{P-2} - 1$; } ★ Set new interval length
 2. if $A > B$ then Propagate_Carry (t, \mathbf{d}); ★ Propagate carry if overflow
 3. Encoder_Renormalization (B, L, t, \mathbf{d}) ★ Output two symbols
 4. return.
-

ALGORITHM 5.27**Procedure Arithmetic_Decoder ($N, M, \mathbf{C}, \mathbf{d}$)**

1. set { $L \leftarrow D^P - 1$; ★ Initialize interval length
 $V = \sum_{n=1}^P D^{P-n} d(n)$; ★ Read P digits of code value
 $t \leftarrow P$; } ★ Initialize symbol counter
 2. for $k = 1$ to N do ★ Decoding loop
 - 2.1. set $s_k = \text{Interval_Selection}(V, L, M, \mathbf{C})$; ★ Decode symbol and update interval
 - 2.2. if $L < D^{P-1}$, then ★ If interval is small enough,
Decoder_Renormalization (V, L, t, \mathbf{d}); ★ renormalize interval
 4. return.
-

ALGORITHM 5.28**Function Interval_Selection (V, L, M, \mathbf{C})**

1. set { $s \leftarrow 0$; $n \leftarrow M$; ★ Initialize bisection search limits
 $X \leftarrow 0$; $Y \leftarrow L$; ★ Initialize bisection search interval
 2. while $n - s > 1$ do ★ Binary search loop
 - 2.1. set { $m \leftarrow \lfloor (s + n)/2 \rfloor$; ★ Compute middle point
 $Z \leftarrow \lfloor [L \cdot C(m)] \cdot D^{-P} \rfloor$; } ★ Compute value at middle point
 - 2.2. if $Z > V$, ★ If new value larger than code value,
then set { $n \leftarrow m$; ★ then update upper limit
 $Y \leftarrow Z$; } ★ else update lower limit
 - else set { $s \leftarrow m$; ★ else update lower limit
 $X \leftarrow Z$; }
 3. set { $V \leftarrow V - X$; ★ Update code value
 $L \leftarrow Y - X$; } ★ Update interval length as difference
 4. return s .
-

length, so that when procedure Encoder_Renormalization is called, it adds the proper two last output symbols to buffer \mathbf{d} . This way, correct decoding does not depend on the value of the symbols that are read by the decoder (depending on register size P) past the last compressed data symbols.

With integer arithmetic the decoder does not have to simultaneously update the interval base and code value (see Algorithms 5.8 and 5.27). Since decoding is always based on the difference

ALGORITHM 5.29

Procedure Decoder_Renormalization (V, L, t, d)

1. while $L < D^{P-1}$ do
 - 1.1. set $\{ t \leftarrow t + 1;$
 - $V \leftarrow (D \cdot V) \bmod D^P + d(t);$
 - $L \leftarrow (D \cdot L) \bmod D^P; \}$
2. return.

★ Renormalization loop
 ★ Increment symbol counter
 ★ Update code value
 ★ Update interval length

$v - b$, we define $V = D^P(v - b)$ and use only V and L while decoding. The only other required change is that we must subtract from V the numbers that would have been added to B .

In Algorithm 5.7 we used sequential search for interval selection during decoding, which in the worst case requires testing $M - 1$ intervals. In Algorithm 5.28 we use bisection [17, 19, 39] for solving (5.16), which requires testing at most $\lceil \log_2(M) \rceil$ intervals (see Sections 5.3.2.3 and 5.3.3.2). Finally, the decoder renormalization is shown in Algorithm 5.29.

ACKNOWLEDGMENTS

I gratefully express my thanks to all the members of the Compression and Multimedia Project at Hewlett-Packard Laboratories, Nelson Chang, Debargha Mukherjee, Ramin Samadani, and especially to I-Jong Lin, who helped me review and correct the text and gave me excellent suggestions.

5.4 REFERENCES

1. Shannon, C. E., 1948. A mathematical theory of communications. *Bell Systems Technical Journal*, Vol. 27, pp. 379–423, July 1948.
2. Huffman, D. A., 1952. A method for construction of minimum redundancy codes. *Proceedings of the IRE*, Vol. 40, pp. 1098–1101.
3. Golomb, S. W., 1966. Run-length encoding. *IEEE Transactions on Information Theory*, Vol. 12, No. 4, pp. 399–401.
4. Gallager, R. G., 1968. *Information Theory and Reliable Communication*, Wiley, New York.
5. Jelinek, F., 1968. *Probabilistic Information Theory*, McGraw-Hill, New York.
6. Pasco, R., 1976. *Source Coding Algorithms for Fast Data Compression*, Ph.D. thesis. Department of Electrical Engineering, Stanford University, Stanford, CA.
7. Rissanen, J. J., 1976. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, Vol. 20, No. 3, pp. 198–203, May 1976.
8. Ziv, J., and A. Lempel, 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343.
9. Gallagher, R. G., 1978. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, Vol. 24, No. 6, pp. 668–674, November 1978.
10. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536.
11. Langdon, G. G., and J. J. Rissanen, 1978. *Method and Means for Arithmetic String Coding*. U.S. Patent 4,122,440, October 1978.
12. Martin, G. N. N., 1979. Range encoding: An algorithm for removing redundancy from a digitized message. In *Video and Data Recording Conference*, Southampton, England, July 1979.

13. Rubin, F., 1979. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory*, Vol. 25, No. 6, pp. 672–675, November 1979.
14. Rissanen, J. J., and G. G. Langdon, 1981. Universal modeling and coding. *IEEE Transactions on Information Theory*, Vol. 27, pp. 12–23, January 1981.
15. Langdon, G. G., and J. J. Rissanen, 1981. Compression of black–white images with arithmetic coding. *IEEE Transactions on Communications*, Vol. 29, pp. 858–867, June 1981.
16. Langdon, G. G., and J. J. Rissanen, 1981. *Method and Means for Arithmetic Coding Utilizing a Reduced Number of Operations*, U.S. Patent 4,286,256, August 1981.
17. Gill, P., M. H. Wright, and W. Murray, 1982. *Practical Optimization*, Academic Press, New York.
18. Langdon, G. G., and J. J. Rissanen, 1982. A simple general binary source code. *IEEE Transactions on Information Theory*, Vol. 28, pp. 800–803, September 1982.
19. Rice, J. R., 1983. *Numerical Methods, Software and Analysis*, McGraw-Hill, New York.
20. Jayant, N. S., and P. Noll, 1984. *Digital Coding of Waveforms*, Prentice-Hall, Englewood Cliffs, NJ.
21. McEliece, R. J., 1984. *The Theory of Information and Coding*, Cambridge Univ. Press, New York.
22. Papoulis, A., 1984. *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York.
23. Langdon, G. G., 1984. An introduction to arithmetic coding. *IBM Journal of Research and Development*, Vol. 28, No. 2, pp. 135–149, March 1984.
24. Knuth, D. E., 1985. Dynamic Huffman coding. *Journal of Algorithms*, Vol. 6, pp. 163–180, June 1985.
25. Witten, I. H., R. M. Neal, and J. G. Cleary, 1987. Arithmetic coding for data compression. *Communications of the ACM*, Vol. 30, No. 6, pp. 520–540, June 1987.
26. Vitter, J. S., 1987. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, Vol. 34, pp. 825–845, October 1987.
27. Pennebaker, W. B., J. L. Mitchell, G. G. Langdon Jr., and R. B. Arps, 1988. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, Vol. 32, No. 6, pp. 717–726, November 1988.
28. Pennebaker, W. B., and J. L. Mitchell, 1988. Probability estimation for the Q-coder. *Journal of IBM Research and Development*, Vol. 32, No. 6, pp. 737–752, November 1988.
29. Bell, T. C., I. H. Witten, and J. Cleary, 1990. *Text Compression*, Prentice Hall, New York, February 1990.
30. Cormen, T. H., C. E. Leiserson, and R. L. Rivest, 1990. *Introduction to Algorithms*, MIT Press, Cambridge, MA.
31. Moffat, A. 1990. Linear time adaptive arithmetic coding. *IEEE Transactions on Information Theory*, Vol. 36, pp. 401–406, March 1990.
32. Cover, T. M., and J. A. Thomas, 1991. *Elements of Information Theory*, Wiley, New York.
33. International Telecommunication Union (ITU). 1992. *Digital Compression and Coding of Continuous-Tone Still Image: Requirements and Guidelines*, ITU-T Recommendation T.81, Geneva, Switzerland.
34. Howard, P. G., and J. S. Vitter, 1992. Practical implementations of arithmetic coding. In *Image and Text Compression* (J. A. Storer, Ed.), Kluwer Academic, Boston, MA.
35. Gersho, A., and R. M. Gray, 1992. *Vector Quantization and Signal Compression*, Kluwer Academic, Boston, MA.
36. Pennebaker, W. B., and J. L. Mitchell, 1992. *JPEG: Still Image Data Compression Standard*, Von Nostrand Reinhold, New York.
37. Howard, P. G., and J. S. Vitter, 1992. Analysis of arithmetic coding for data compression. *Information Procedures and Management*, Vol. 28, No. 6, pp. 749–764.
38. International Telecommunication Union (ITU). 1993. *Progressive Bi-Level Image Compression*, ITU-T Recommendation T.82, Geneva, Switzerland.
39. Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, 1993. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press, Cambridge, UK.
40. Shapiro, J. M., 1993. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, Vol. 41, pp. 3445–3462, December 1993.
41. Burrows, M., and D. J. Wheeler, 1994. *A Block-Sorting Lossless Data Compression Algorithm*, Technical Report, Digital Equipment Corp., Palo Alto, CA.
42. Fenwick, P. M., 1994. A new data structure for cumulative frequency tables. *Software Practices and Experiments*, Vol. 24, pp. 327–336, March 1994.

43. Said, A., and W. A. Pearlman, 1995. Reduced-complexity waveform coding via alphabet partitioning. *Proceedings of the IEEE International Symposium on Information Theory*, Whistler, Canada, p. 373, September 1995.
44. International Telecommunication Union (ITU), 1996. *Standardization of Group 3 Facsimile Terminals for Document Transmission*, ITU-T Recommendation T.4, Geneva, Switzerland.
45. Weinberger, M. J., J. J. Rissanen, and R. B. Arps, 1996. Applications of universal context modeling to lossless compression of gray-scale images. *IEEE Transactions on Image Processing*, Vol. 5, pp. 575–586, April 1996.
46. Wu, X., and N. Memon, 1996. CALIC—A context based adaptive lossless image codec. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Vol. 4, pp. 1890–1893, May 1996.
47. Said, A., and W. A. Pearlman, 1996. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuitry Systems and Video Technology*, Vol. 6, pp. 243–250, June 1996.
48. Bhaskaran, V., and K. Konstantinides, 1997. *Image and Video Compression Standards: Algorithms and Architectures*, Kluwer Academic, Dordrecht/Norwell, MA.
49. LoPresto, S., K. Ramchandran, and M. T. Orchard, 1997. Image coding based on mixture modeling of wavelet coefficients and a fast estimation–quantization framework. *Data Compression Conference*, Snowbird, UT, pp. 221–230, March 1997.
50. Moffat, A., 1997. Critique of the paper “Novel design of arithmetic coding for data compression.” *IEE Proceedings—Computer and Digital Technology*, Vol. 144, pp. 394–396, November 1997.
51. Moffat, A., R. M. Neal, and I. H. Witten, 1998. Arithmetic coding revisited. *ACM Transactions on Information Systems*, Vol. 16, No. 3, pp. 256–294.
52. Schindler, M., 1998. A fast renormalisation for arithmetic coding. *Proceedings of the IEEE Data Compression Conference*.
53. Witten, I. H., A. Moffat, and T. C. Bell, 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd ed., Morgan Kaufmann, San Mateo, CA.
54. Wu, P-C., 1999. “A byte-wise normalization method in arithmetic coding.” *Softw. Pract. Exper.*, Vol. 29, pp. 299–309, April 1999.
55. Sayood, K., 2000. *Introduction to Data Compression*, 2nd ed., Morgan Kaufmann, San Mateo, CA.
56. Salomon, D., 2000. *Data Compression*, 2nd ed., Springer Verlag, New York.
57. Sodagar, I., B-B. Chai, and J. Wus, 2000. A new error resilience technique for image compression using arithmetic coding. *Proceedings of the International Conference on Acoustics, Speech and Signal Processes*, Vol. 4, pp. 2127–2130, June 2000.
58. Texas Instruments Inc., 2000. *TMS320C6000 CPU and Instruction Set Reference Guide*, Literature No. SPRU189F, Dallas, TX.
59. International Business Machines Corp. 2001. *PowerPC 750CX/CXe RISC Microprocessor User’s Manual* (preliminary edition), Hopewell Junction, NY.
60. Intel Corp., 2001. *Intel Pentium 4 Processor Optimization*, Reference Manual 248966, Santa Clara, CA.
61. Sun Microsystems, Inc., 2001. *UltraSPARC III Technical Highlights*, Palo Alto, CA.

Dictionary-Based Data Compression: An Algorithmic Perspective

S. CENK SAHINALP
NASIR M. RAJPOOT

6.1 INTRODUCTION

Dictionary-based methods are among the most popular forms of lossless data compression. Given an input string of characters from a constant size alphabet A (e.g., ascii), a dictionary-based compression algorithm partitions the input string S into non-overlapping substrings which are called *phrases* and replaces each phrase with corresponding bit strings which are called *codewords*; the *dictionary* is this function which maps phrases to codewords. The fundamental task of a dictionary-based method is to partition the input S into phrases whose corresponding codewords have the smallest total bit-length.

Dictionary-based compression is sometimes referred to as *compression with textual substitution*. Textual substitution methods can naturally be classified into four categories according to whether they allow *fixed-* or *variable-length* phrases and *fixed-* and *variable-length* codewords. For us, dictionary-based compression will mean only textual substitution with variable-length phrases; the codewords on the other hand can be fixed or variable length. Note that in compression methods with fixed-length phrases (such as Huffman coding or some of the context sensitive methods), partitioning ceases to be an issue and dictionary construction becomes the main concern; better compression is achieved by replacing more frequent phrases with shorter codewords.

A dictionary-based compression method comprises two relatively independent stages. The first stage is the *dictionary construction*: the set of (potential) substrings of the input are determined as phrases and are given unique codewords. The second stage is *parsing*: the input string is partitioned into phrases which are then replaced with corresponding codewords.

Similar to other compression schemes, dictionary-based compression methods need to satisfy the following two constraints (see Section 3.1 in [22] for a similar set of constraints):

1. **Generality.** Every input string (of interest to the application in question) should be “compressible”; i.e., it should be possible for the algorithm to process every input string. It could be the case that the compressed representation is bit-wise longer than the original input.
2. **Unique decompressibility.** Every compressed string should be uniquely decompressible; i.e., the compression algorithm needs to have a corresponding decompression algorithm which can uniquely obtain the original representation of any input string from its compressed representation.

Thus the first goal of a dictionary compression algorithm is to maximize the compression achieved while satisfying the above constraints. The achieved compression can be measured in terms of the *compression ratio*: the ratio of the number of bits in the original representation of the input and the number of bits in its compressed representation. (An alternative measure is the *compression rate* which is defined as $1 - \text{compression ratio}$.) A secondary goal is to be as efficient as possible in terms of time and space utilization. Clearly, the compression ratio is a function of both the dictionary construction method and the parsing method employed. The overall efficiency of a compression algorithm is usually determined by its efficiency in searching for suitable phrases in the dictionary and hence is related to the *data structure* employed for representing the dictionary. In the following sections, we elaborate on these three main issues, namely, dictionary construction, parsing, and the data structure employed. We also discuss additional constraints on compressed representations of strings such as fault tolerance, searchability, and indexability. Finally, we provide efficiency/hardness results on select compression/parsing methods and discuss possible uses of multiple dictionaries, as well as compression methods inspired by dictionary methods such as anti-dictionaries. We also present several open problems during our discussions.

6.2 DICTIONARY CONSTRUCTION: STATIC VERSUS DYNAMIC

Dictionary construction methods are usually classified into three basic families: *static*, *semidynamic*, and *dynamic* (some texts prefer to use the term *adaptive* instead of dynamic).

A static dictionary method uses the same dictionary for all input strings; thus such dictionaries are used only in specific applications where the input strings of interest contain many common phrases. A semidynamic dictionary method builds a “static” dictionary by inspecting the input string as a whole; the compression (i.e., parsing) is performed only after the dictionary is constructed in full. For more general applications, dynamic methods are typically used. A dynamic dictionary method updates its dictionary as it parses and compresses the input string; thus parsing and dictionary construction steps usually interleave.

In the following sections we will first motivate dictionary compression by analyzing the properties of static dictionaries, then we will discuss parsing issues for static dictionaries as well as dynamic dictionaries, and finally we present some of the well-known approaches to dynamic dictionary construction.

6.2.1 Static Dictionary Methods

As we indicated earlier, a static dictionary is simply a set of substrings from the input alphabet with corresponding codewords. Ideally the dictionary should consist of phrases common to input strings which are typically encountered in the application domain. Clearly the dictionary used needs to be available to both the compression algorithm and its corresponding decompression algorithm, and, as other dictionary methods, should satisfy the generality and the unique decompressibility constraints.

The generality constraint can be computationally verified for static dictionaries via the following lemma.

Lemma 6.1. *Any dictionary D consisting of phrases from an input alphabet A satisfies the generality constraint if and only if for any given string S of characters from A there exists a prefix S' of S which is a phrase in D .*

Proof. If there exists a string S for which no prefix is a phrase in the dictionary, clearly S is not compressible by D . If for any string S there exists at least one prefix S' which is a phrase in D , then S must be compressible by D : Let $S = S'S''$; by the initial assumption, the string S'' must have a prefix which is a phrase in D , hence the proof follows. ■

The above condition which is equivalent to the generality constraint can be verified in $O(|D|)$ time and space, where $|D|$ denotes the total number of characters in all phrases of D .

Unfortunately we are not aware of any similar condition that can efficiently check whether a dictionary satisfies the unique decompressibility constraint. Thus our first open problem follows:

Open Problem 6.1. *Analyze computationally verifiable conditions that are necessary and sufficient for a dictionary to satisfy the generality and the unique decompressibility constraints.*

Nevertheless it is easy to construct dictionaries satisfying the unique decompressibility constraint by the help of the following sufficiency lemma.

Lemma 6.2. *Consider a dictionary D consisting of phrases with corresponding binary substrings as codewords; if no codeword in D is the prefix of another, then D satisfies the unique decompressibility property.*

Proof. If no codeword of D is a prefix of another, then given a binary string S obtained by a compression algorithm that uses D , there is a unique prefix which can be extracted from S and replaced by a corresponding phrase regardless of the parsing method used in the compression algorithm. The proof follows by iteratively applying this argument. ■

6.2.2 Parsing Issues

As mentioned earlier the ultimate goal of parsing is to partition the input string into phrases whose corresponding codewords have the smallest possible total bitwise length. Although most parsing methods can be employed by both static and dynamic dictionary methods, it is more informative to introduce them in the context of static dictionaries for which optimality results in terms of compression and performance can be given. We will only get to the specific details of parsing for dynamic dictionaries during our discussion on these methods.

For static dictionary methods, the optimal parsing of any input string can be achieved by a two-pass dynamic programming algorithm, which first reads the input from left to right to determine the “best” parsing, and then from right to left to replace the phrases obtained with respective codewords, as follows.

Let $D = \{d_1, \dots, d_k\}$ be the dictionary and its phrases and let c_i be the codeword of phrase d_i ; the lengths of d_i and c_i are respectively represented as $|d_i|$ and $|c_i|$. Let the input string be $S = s_1, \dots, s_n$. The algorithm constructs an array $A[1 : n]$ where $A[i]$ is the length of the optimal compression (i.e., its shortest possible representation by using D) of s_1, \dots, s_i . It also constructs another array $B[1 : n]$, where $B[i]$ is the starting location of the last phrase replaced by a codeword in the optimal compression of s_1, \dots, s_i .

Set $A[0] = 0$ and $A[i] = \infty$ for all $i \neq 0$. The algorithm iteratively computes $A[i]$ by using $A[1, \dots, i - 1]$ as follows. For each i , it iteratively checks for $j = 1, \dots, k$ whether the phrase

d_j matches $s_{i-|d_j|+1}, s_i$ and replaces the current value of $A[i]$ with $A[i - |d_j|] + |c_j|$ and $B[i]$ with $i - |d_j|$ if the latter is smaller. Once $A[n]$ is computed, all that is needed is to follow the pointers of B iteratively, starting from $B[n]$, while replacing the phrases with their corresponding codewords.

Lemma 6.3. *The above dynamic programming routine obtains the shortest possible representation of any given string S when it is compressed with static dictionary D .*

Proof. Assume the contrary; then there should exist at least one $A[i]$, which does not represent the size of the shortest compressed representation of s_1, \dots, s_i . Consider the smallest such i (i.e., the leftmost position). The dynamic programming method assigns to $A[i]$ the smallest possible $A[i - |d_j| + 1] + |d_j|$ among all phrases d_j which match a suffix of s_1, \dots, s_i . Thus if $A[i]$ is incorrectly computed, then there should exist at least one $h < i$ for which $A[h]$ is incorrectly computed, contradicting the assumption that $A[i]$ is the leftmost incorrectly computed entry of A . ■

The naive implementation of the above dynamic programming routine runs in $O(n \cdot |D|)$ time (where $|D|$ is the total size of all phrases in the dictionary). There is a more efficient $O(n \cdot \max_i |d_i|)$ time algorithm, where $\max_i |d_i|$ is the size of the longest phrase in the dictionary by the use of a simple trie data structure, which we will discuss later in the chapter. Nevertheless, because many (especially communication related) applications require an on-line processing of the input string and thus are not suitable for two-pass parsing and because there is an ever-increasing demand to compress the input faster, such a dynamic programming approach is often not used in practice. Rather, available methods usually employ specialized dictionaries for which there are on-line parsing methods that achieve optimality or near optimality in a single pass. Such schemes make parsing decisions based on local information about the input string and cannot afford to look ahead or look back for more than a few steps.

Among on-line parsing methods, by far the most popular one (in both static and dynamic methods) is the greedy strategy: While compressing the input from left to right, the greedy method simply parses (and replaces with the corresponding codeword) the longest phrase that matches a prefix of the uncompressed portion of the input. This approach requires no more than a single pass over the input string and uses no information about the “past” or “distant future” and thus the associated delay is limited. Moreover, it enables one to compress a string of size n in optimal $O(n)$ time (optimality follows from the fact that each one of the n characters needs to be read and processed) by the use of a trie data structure to represent the dictionary. The space requirement of this data structure is $O(|D|)$, which is again optimal. Greedy strategy parses any input string to the minimum number of codewords possible (and hence achieves the best possible compression when all codewords are of same length) if the dictionary is suffix complete [3]: A given dictionary D is suffix complete if for every phrase in D , all its suffixes are also phrases in D .

Suffix-complete dictionaries are not very common in practice; one exception is the Lempel-Ziv 1977 method [30], which will be described in more detail in the discussion on dynamic dictionaries. Most practical dictionaries (which are generally dynamic) are prefix complete; i.e., for each phrase in D all its prefixes are also phrases in D . For such dictionaries, compression with greedy parsing can be far from optimal on certain input strings. However, by a slight modification to the greedy rule, it is possible to obtain optimality for prefix-complete dictionaries: *Greedy parsing with one-step lookahead* achieves the smallest number of codewords on any input when used with a prefix-complete dictionary. This variant of greedy parsing is dubbed flexible greedy parsing in [17, 29], which provides optimality results for dynamic dictionaries; optimality results for static dictionaries were provided earlier in [11].

Flexible greedy parsing (similar to the standard greedy parsing) reads the input characters from left to right. In each step, it finds the two phrases whose concatenation matches the longest prefix

of the uncompressed portion of the input. It then replaces the first phrase’s occurrence in the input with the corresponding codeword and iterates.

It is possible to implement the flexible greedy parsing in optimal $O(n)$ time via a trie–reverse trie pair in space proportional to the size of the compressed version of the string [17, 29], which is again optimal. It may be possible to improve compression rates while maintaining $O(n)$ time performance by employing dictionaries that can utilize greedy parsing methods with more than a single-step lookahead. In fact the general dynamic programming approach described earlier can be seen as a greedy method with an unlimited number of lookaheads. Unfortunately we are not aware of any compression algorithms in the literature that exploit such an approach.

6.2.3 Semidynamic and Dynamic Dictionary Methods

A static dictionary may also be precomputed by inspecting the input string to be compressed and inserting in the dictionary some of its frequently occurring substrings as phrases. In this context, the dictionary itself (which can also be compressed via several means) should be considered as part of the compressed version of the input string. This *semidynamic* approach, thus, should aim to construct the “best” static dictionary suitable for one or more documents to be compressed. Once the dictionary is constructed via a “training” input string, it is possible to use it later for input strings from the same application domain.

A rigorous treatment by Storer [22] classifies dynamic dictionary methods into four basic *macro schemes* as follows.

All semidynamic dictionary methods can be investigated under the *external pointer macro scheme* (EPM). In this macro scheme, the compression of an input string S is achieved by constructing a “dictionary string” T and then replacing substrings of S with pointers to identical substrings in T . Thus the compressed representation of S is the tuple (T, S') , where S' is the string of pointers to substrings of T , and characters from the original alphabet. Note that it is possible to compress T as well by replacing substrings of T with pointers to other occurrences of the same substring within T .

In a *compressed pointer macro scheme* (CPM), the compressed representation S' of input string S can be decompressed by considering the tuple (S', S') as one obtained by an EPM scheme and thus treating S' as its own dictionary.

In an *original pointer macro scheme* (OPM), the compressed representation S' of input S can be decompressed by replacing in S' each pointer of any given length k , by k pointers of length 1, and then treating it as a CPM scheme.

In an *original external pointer macro scheme* (OEPM), the compressed form of input S consists of a tuple (T', S') where T' is the OPM compressed form of a dictionary T , which is used to EPM compress S .

Storer gives several sufficient conditions for specific implementations of these schemes so that they satisfy the generality and unique decompressibility constraints. He also provides hardness results related to the computational complexity of obtaining optimal compression under specific schemes, in particular the CPM scheme, which is especially important for practical purposes. We will come back to some of these hardness results later in this chapter.

6.2.3.1 Dynamic Dictionary Methods

Almost all practical dynamic dictionary methods are specific implementations of the CPM. At the intuitive level, a dynamic dictionary method aims to “learn” and adaptively insert in the dictionary substrings that are frequently observed in the input. It is possible to insert in the dictionary certain substrings which are not observed in the input; we will give examples of such schemes later in our discussions (on approximate dictionaries, anti-dictionaries, etc.). For now, we focus only on

dictionaries which consist of substrings that are present in the input string. In such schemes, a codeword that replaces a substring of the input can be seen as a *pointer* to another occurrence of the same substring. Thus dynamic dictionary methods can naturally be classified into two categories: *unidirectional*, in which all pointers have the same direction (e.g., they always point left), and *bidirectional*, in which pointers can point both left and right. We discuss unidirectional and bidirectional algorithms separately in the following sections.

6.2.3.2 Unidirectional Methods

Most practical dictionary methods are unidirectional. As mentioned above, these methods use codewords that all point to the same direction (without loss of generality we will assume that this direction is *left*). Thus such methods usually enable one to compress a given input string iteratively, in a *single pass*: Suppose that a prefix of the input is already compressed; then all that is needed to do is to find the “best” prefix of the uncompressed portion of the input which can be replaced by a pointer to an occurrence in the compressed portion of the input. One can readily observe that all *unidirectional* methods enable decompression in a single pass, which is a desired feature for applications in which computational power and memory of the decompressing party (such as in mobile communications) are much lower than those of the sending party. Thus all unidirectional methods that we are aware of are actually single-pass methods in terms of both compression and decompression.

Single-pass methods are especially popular due to the fact that they usually have *on-line* implementations where the goal is to limit the delay between reading a character and outputting a codeword representing the character (within a phrase). Such implementations are crucial in communication applications, which probably explains why pretty much all single-pass methods are on-line implementable. Because all unidirectional methods in the literature are single pass, and all single-pass methods have on-line methods, these terms are sometimes used in place of the others.

In the next few pages we discuss some of the best known unidirectional/single-pass/on-line methods by describing their dictionary construction and parsing methods and the synchronization between these two methods. We also provide some compressibility results, asymptotic and nonasymptotic.

6.2.3.2.1 Some History

The general idea of compression by textual substitution is usually attributed to a 1967 paper by White [28], which suggests a generic unidirectional scheme as a concluding remark. This suggestion was realized in a powerful method by Ziv and Lempel in 1977 [30], which we describe in detail later in this section. Their seminal paper proves that their algorithm is asymptotically optimal for input strings which are generated by an *ergodic* source (a very general class of probabilistic sources). The first algorithmic realization of this method required a running time of $O(n^2)$, where n is the number of characters in the input. One year later, Ziv and Lempel [31] came up with an alternative method with similar asymptotic properties with a very straightforward $O(n)$ time algorithmic realization. In 1981, Rodeh *et al.* [19] demonstrated that it was possible to realize the first method in $O(n)$ time as well. Later it was shown independently in [12, 14] and in [20] that the second algorithm gets closer to asymptotic “optimality” faster than the first algorithm.

The Ziv and Lempel methods were popularized by a version of their second algorithm obtained by Welch [26, 27]. Interestingly, it was Welch who obtained the first patent for the Lempel–Ziv-based approaches. This version became the basis of many popular programs including the GIF image compression format, UNIX `compress` and `compact` programs and became part of the popular V42 .bis modem compression standard. (A newer modem compression standard, V44, also uses an algorithm that can be thought as a combination of the two Ziv–Lempel methods.)

6.2.3.2.2 Lempel-Ziv-77 Method Among the popular data compression methods, the Lempel-Ziv-77 method (denoted LZ77) deserves special attention. Although it is not the first dynamic dictionary method proposed, it clearly is the one which popularized the general approach.

Before presenting the original version of the LZ77 algorithm, we describe a simpler variant by Storer and Szymanski [21] (denoted LZSS). The parsing method employed by the LZSS is greedy (with no lookahead). The dictionary, at each step, simply consists of (1) all single-character substrings and (2) all substrings of the input which start in the already compressed prefix of the input and finish before the end of the phrase to be parsed. Each codeword consists of a bit flag that denotes whether the codeword is of type (1) or (2). Type (1) codewords (of single-character phrases) simply are composed of the respective character; type (2) codewords are composed of two entries: (i) a pointer to the start location of the last occurrence of the phrase and (ii) the length of the phrase. Both the pointer and the length information can be represented by a fixed number of bits or can be represented through Elias (universal) coding. (For brevity we omit some details of the LZSS coding method.) (Universal coding is described in Chapter 3.)

In the original LZ77 method, all codewords are identical in composition. They are composed of the following three entries: (i) a pointer to the start location of the last occurrence of the phrase (set to 0 if no such occurrence exists); (ii) the length of the phrase (set to 0 if no such occurrence exists); and (iii) the first character of the uncompressed portion of the input; the next step of parsing is performed after skipping this character.

A first glance at the LZ77 algorithm may give the impression that the richness of the dictionary may degenerate the compression ratio achieved and running time of the algorithm; two papers show that this is not the case: (i) Ziv and Lempel [30] were able to demonstrate that if an *ergodic* source generates the input string, as the input gets larger, the compression ratio achieved by the LZ77 algorithm gets arbitrarily close to the entropy of the source. Thus, for ergodic sources, the LZ77 algorithm is *asymptotically optimal*. Our focus in this presentation is more on the combinatorial issues; thus we do not provide the details of this result here. (ii) Rodeh *et al.* [19] later showed that an on-line version of the *suffix-tree* data structure [16] enables one to implement the LZ77 algorithm in time and space linear with the input size. We will give further details of this implementation in the Data Structures in Dictionary Compression section.

One final observation is that the LZ77 dictionary construction method is suffix complete (all suffixes of a phrase are by definition phrases) and thus when it is used for compressing a given input, the greedy parsing should achieve the minimum number of phrases possible. Thus, if the lengths of the codewords are all equal, the greedy parsing achieves the best compression possible by the LZ77 dictionary construction. We are not aware of any on-line or off-line parsing scheme that achieves optimality when the LZ77 dictionary is in use under any constraint on the codewords other than being of equal length.

6.2.3.2.3 LZ77 Variants There are many variants of the LZ77 algorithm; for example, a sliding window version does not allow pointers of size more than a user-defined bound. There are also mixed schemes which normally allow only fixed-size windows but may also tolerate longer matches when opportunities for large replacements arise [15]. Other versions try to compress again the sequence of pointer and/or phrase length entries of the codewords separately, by other means of compression (see, for example, [1]), such as Huffman coding (see [2]) and arithmetic coding. Such variants, as well as the original algorithm, have been used in many applications and provided the basis for many popular programs such as all the `zip` variants (e.g., `gzip`, `winzip`, `pkzip`), `PNG` (an image compression format), `arj` (an archive compression format), and `LHarc` (another archive compression program).

6.2.3.2.4 Lempel–Ziv–78 Method The second algorithm by Ziv and Lempel (denoted LZ78) was popularized by a variant reported by Welch [27] and is known as the LZW algorithm. The LZW algorithm is simpler and thus is described first: It uses greedy parsing as per the LZ77 algorithm. The difference is in dictionary construction. The dictionary initially consists only of single-character substrings. After each parsing step, the phrase parsed is concatenated with the first character of the uncompressed portion of the input and is inserted in the dictionary as a new phrase. This phrase is assigned $|D| + 1$ as a codeword (where $|D|$ is the number of phrases in the dictionary). At any given step of the algorithm each phrase is represented by $\lceil \log_2 |D| \rceil$ bits.

The original LZ78 algorithm again uses a variant of greedy parsing. At each step of the algorithm, the longest prefix of the uncompressed portion of the input is parsed and replaced with a codeword; the following character is skipped uncompressed to iteratively continue the compression process. The dictionary at a given step includes all parsed phrases concatenated with the single characters following them, which are left uncompressed.

In their seminal paper, Ziv and Lempel [31] demonstrated that this algorithm also achieves asymptotic optimality when the input is generated by an ergodic source. The advantages of this algorithm are simplicity and the newly discovered property that the compression ratio achieved by the LZ78 algorithm and the LZW variant approaches the *entropy* (a measure of compressibility, which translates into the best asymptotic compression possible by any algorithm) of an ergodic source generating the input faster than that of the LZ77 (see [12, 14, 20]).

6.2.3.2.5 LZ78 Variants The most popular variants of the LZ78 scheme include those used in the UNIX compress program, the GIF image compression format, and the V42.bis modem standard, which all simply use the LZW method with minor modifications on the dictionary maintenance (see discussion below). Other variants focus on modifying the dictionary construction and parsing method employed.

6.2.3.2.6 Modifications in Dictionary Construction There are many ways to update the dictionary in a way similar to the LZ78 scheme. One immediate modification [9] considers extending the longest parsed phrase by some fixed k characters as well as $k - 1, \dots, 1$ characters. This is similar to an earlier modification by Miller and Wegman [18] which concatenates the longest parsed phrase with the following longest phrase parsable and inserts it in the dictionary. A combination of these two ideas was considered by Storer so that all prefixes of each phrase inserted in the dictionary by the Miller–Wegman method are inserted in the dictionary as well (see [22]). Later, Horspool considered updating the dictionary by parsing the longest string greedily at each step and inserting in the dictionary with a single-character extension as in the case of LZ78, but before outputting the corresponding codeword, checking out if one can reach further if parsing restarts not at the immediate next character but a few characters back [10]. Because the LZ78 dictionary is prefix complete, one can rather replace with its codeword the prefix which gives the furthest reach in the next step.

6.2.3.2.7 Modifications in Parsing It was shown [17, 29] that flexible greedy parsing, when performed independent of the dictionary construction (which, on any string, constructs the exact same dictionary as that obtained by the LZ78 method), achieves the minimum number of codewords possible by the dynamic dictionary constructed through the LZ78 method. In fact, this optimality result is valid for any dynamic dictionary construction method which maintains the prefix completeness property at all times. It is also shown that one-step lookahead greedy parsing can be implemented within the same time and space bounds of the original LZ78 algorithm and

that the corresponding decompression algorithm also correctly decompresses data compressed by the original LZ78 algorithm.

6.2.3.2.8 Dictionary Maintenance in Dynamic Methods Another important issue in implementations of Lempel–Ziv methods and other dictionary-based algorithms is how to handle memory limitations on the dictionary. One simple solution is to put a bound on the size of the dictionary and start building the dictionary again once it gets filled up. Other methods for cleaning up the dictionary once it is full include deleting the least recently used (LRU), least frequently used (LFU), swap, or freeze heuristics (see, for example, [6], and check [22] for details).

The freeze heuristic simply stops inserting new phrases to the dictionary once it is full and compresses the rest of the input by using the dictionary without any changes.

The LRU heuristic deletes the least recently used phrase/codeword pair from the dictionary. The codeword that belonged to the deleted phrase could be used for a new phrase inserted in the next step. To implement this heuristic, one needs to keep time stamps for every phrase which should be updated every time it is accessed. The time stamps could be maintained in a priority queue, which can efficiently provide which phrase is to be deleted once the dictionary is full.¹ One obvious drawback of the LRU heuristic is the need for extra memory for storing the priority queue.

The LFU heuristic deletes the least frequently used phrase/codeword pair. It is very hard to implement the LFU heuristic if the frequency of a phrase is defined to be the number of times it is accessed, normalized by the number of accesses to the whole dictionary since its insertion; this may require updating the frequency information of every phrase at each step. Another version, in which the normalization is done by the total number of accesses to the dictionary since the compression started, can be implemented through priority queues within the space and time bounds for the LRU heuristic.

A natural heuristic is to simply delete the dictionary when it gets full and start building it from scratch. An alternative for avoiding “forgetting the past” altogether is provided by the swap heuristic, which constructs a primary dictionary while constructing an auxiliary dictionary simultaneously. Only the primary dictionary is used for compression purposes. When the primary dictionary is full, the algorithm deletes the primary dictionary and swaps it with the auxiliary one.

6.2.3.3 Bidirectional Methods

Because bidirectional methods allow codewords that point to “future” occurrences of phrases, they are off-line by nature and are usually slower than their on-line siblings. The benefit is usually improved compression via more careful parsing.

A very important issue in bidirectional parsing is decompressibility: It should be possible to start out with the uncompressed portions of the compressed string and deduce the compressed portions without ambiguity. One can impose a number of restrictions on a bidirectional method to ensure decompressibility (see Section 5.1 in [22] for a similar set of restrictions). One natural restriction is allowing no circular dependencies of size 2; under this restriction, there cannot exist two pointers each of which points to (portions of) substrings represented by the other. This can be generalized to a restriction on circular dependencies of size k , under which one cannot start from a pointer and reach one of its substrings by following a chain of k or more pointers.

¹ For n phrases, such a priority queue can be implemented in $O(n)$ space, $\Omega(\log n)$ time for at least one of the following operations: (i) updating a phrase; (ii) deleting the least recently used phrase; or (iii) inserting a new phrase.

6.3 EXTENSIONS OF DICTIONARY METHODS FOR COMPRESSING BIOMOLECULAR SEQUENCES

Recently, there has been a dramatic increase in interest in processing biomolecular sequences such as DNA, RNA, and protein sequences. Along with this surge in interest comes a massive amount of biomolecular information that needs to be stored and transmitted across the networks efficiently, and thus arises the crucial need for compression. These sequences may typically contain information about proteins or genes or the complete genetic information about a living organism. The human genome, for example, contains the sequence information of all 23 pairs of human chromosomes and spreads across approximately three billion characters from the alphabet {A, C, G, T} representing the four nucleotides. The protein sequences, on the other hand, are much shorter (with an average length of approximately 300) and are composed of symbols from the 20-character amino acid alphabet. Also of interest are structured and semistructured Web documents related to the health sciences in HTML, XML, and the newly emerging BIOXML formats.

The first studies aimed toward understanding the composition of the human genome seemed to suggest that the distribution of nucleotides and amino acids was uniformly random, which would imply that extracting some non-negligible compression out of (portions of) the genome would be a very difficult task. Moreover, the well-known dictionary compression programs (such as `gzip` and `compress`) usually achieve negative compression rates on such sequences. Here we describe two dictionary-based compression methods which have been shown to produce not only positive compression rates but also reasonable amounts of compression: *biocompress* and *GenCompress*.

6.3.1 The *Biocompress* Program

Grumbach and Tahi [7] exploit the fact that the human genome consists of many exact *tandem* repeats and *complementary palindromes*. A tandem repeat is a concatenation of many copies of a short sequence. A complementary palindrome, on the other hand, is a sequence of length n where the i th nucleotide is complementary to the $(n - i)$ th nucleotide (nucleotide pairs A–T and G–C are complementary) and thus lead to *hairpin* structures. These structures make dictionary-based compression possible, especially when reversals and complements of phrases are allowed. The *biocompress* and *biocompress-2* programs attempt to achieve this via an LZ77-based approach by maintaining all observed sequences of size smaller than a user-determined bound on a complete 4-ary tree. The preprocessed tree enables one to search for reversed complements of phrases in addition to standard exact searches. The *biocompress-2* program attempts to achieve further compression via a second-order arithmetic coder to encode areas of the input that contain no repetitions. On tandem repeat- and reverse palindrome-rich regions of the human genome, these programs are reported to achieve compression rates of up to 32%.

6.3.2 The *GenCompress* Program

A more recent work by Chen *et al.* [4] makes use of the fact that the human genome is rich with sequence duplications within a small percentage of edit differences. These differences are usually single nucleotide replacements but can also be in the form of omissions or additions of nucleotides. The *GenCompress* program is thus based on finding approximate matches of the uncompressed portions of the input in the already compressed part of the genome sequence of interest in the LZ77 fashion. It finds the longest prefix of the uncompressed portion of the input which occurs in the compressed portion with a single-edit difference and replaces it with an appropriate codeword

that encodes not only the location and the length of the match but also the position and character value of the difference. The *GenCompress* program achieves compression rates of up to 44% on sequences mentioned in the context of *biocompress*.

6.4 DATA STRUCTURES IN DICTIONARY COMPRESSION

Data structures and their efficient implementations play a key role in determining the time and space complexity of a data compression algorithm. A data structure for implementing a dictionary should be able to store common phrases in a space-efficient way. It should also allow the efficient performance of the following operations: *insertion* of a new phrase into the dictionary, *searching* for a given substring and returning its corresponding codeword if it is present in the dictionary as a phrase, and occasional *deletion* of an already existing phrase from the dictionary. In some dictionary-based compression methods, it is also required that the data structure supports two additional operations, namely, *extend* and *contract* (see, for example, [29]). Given a phrase S in the dictionary D , the operation $\text{extend}(S, a)$ finds out whether the concatenation of S and a is a phrase in D , whereas the operation $\text{contract}(S)$ finds out whether the suffix $S[2 : |S|]$ already exists in the dictionary. We will discuss a number of fundamental data structures commonly employed in the dictionary-based data compression methods we described. We start with the description of the most basic trie data structure followed by suffix trees [16], trie-reverse trie pairs [17], and hash tables particularly for efficient implementations of Karp–Rabin fingerprints [13].

6.4.1 Tries and Compact Tries

A *trie* T is a labeled tree in which (i) each edge of a trie is labeled with a single character from the input alphabet Σ ; (ii) sibling edges have different labels; (iii) each phrase P is represented by a node n_P in the trie; the phrase P itself can be obtained by concatenating the labels of the edges on the path from the root to node n_P . Clearly each prefix $P[1 : i]$ of any given phrase P corresponds to the i th internal node on the path from the root to n_P . Both the insertion of a substring P in a trie T and the searching of P in T can be trivially performed in $O(|P|)$ time when $|\Sigma|$ is a small constant. For larger alphabets a slightly more involved implementation can perform searches and insertions in $O(|S| \log |\Sigma|)$ time. A *compact trie* improves the space efficiency of a trie by allowing each edge to be labeled by more than a single character while making sure that the first characters of the labels of sibling edges are different.

6.4.2 Suffix Trees

The *suffix tree* of a given string S is the compact trie $T(S)$ of all the suffixes of S . Each leaf in $T(S)$ thus represents a suffix and so a position in S . The simple observation that makes the suffix tree a powerful data structure for a string search is that any substring of S is a prefix of a suffix of S . Thus to find *all* occurrences of (i.e., substrings that are identical to) a given pattern P in S , one needs to start at the root of $T(S)$ and follow the path labeled with characters of P to reach a node r . The leaves (each of which corresponds to a position in S) in the subtree of r give the positions of the occurrences of P in S .

Suffix trees were introduced by Weiner [25]. Since then various algorithms have been presented in the literature to construct a suffix tree for a given string S in time and space linear with $|S|$. These algorithms include [5, 16, 24] as well as [8, 23], which were originally designed for parallel machine models but provide optimal serial implementations. The original linear time algorithm

of McCreight [16] was not designed for an on-line construction of the suffix tree. Rodeh *et al.* [19] described an on-line version of the McCreight method for using suffix trees for an efficient implementation of the LZ77 method. With this approach it is possible to find the longest prefix match of the uncompressed portion of the input S in its already compressed portion in time proportional to the size of the prefix matched. Because the construction of the suffix tree takes an overall time of $O(|S|)$, the total time for executing the LZ77 algorithm on S is optimal $O(|S|)$.

6.4.3 Trie–Reverse Trie Pairs

For one-pass dictionary-based data compression algorithms employing greedy parsing, tries provide a simple and powerful mechanism for finding the longest prefix matches. If one wants to incorporate one or more steps of lookahead, however, tries do not necessarily provide the most efficient method for dictionary maintenance. In particular, greedy parsing with one-step lookahead requires finding a prefix match which provides the farthest reach in the next step. This is possible if one can perform not only prefix extensions but also suffix contractions during the search.

A trie–reverse trie pair [17, 29] is composed of the (compressed) trie T of all phrases in the dictionary, a separate trie T^R of *reverses* of all phrases in the dictionary (the reverse of string $S[1], S[2], \dots, S[k]$ is $S[k], S[k-1], \dots, S[1]$), and pointers between the pair of nodes r and r^R that correspond to the phrase P in T and its reverse P^R in T^R , respectively.

As in the case of a simple trie, the insertion of a phrase P into this data structure takes $O(|S|)$ time. Given a dictionary phrase P , and the node r which represents P in T , one can find out whether the substring obtained by extending P with any character a is in D , by checking out if there is an edge from r with corresponding character a . Thus such an *extension* operation can be done in $O(1)$ time. To find out whether the suffix $P[2 : |P|]$ of P is a phrase in the dictionary, one needs to simply follow the pointer from r to r^R (which will effectively reverse the phrase P) and move one character up in T^R and check whether the reverse of the substring that corresponds to this node ($P[2 : |P|]$) is a phrase in D . Clearly this *contraction* operation can be performed in $O(1)$ time.

One can perform successive extension and contraction operations to find the two consecutive prefixes that reach the farthest as follows. Let P^0 be the longest prefix of S^U , the uncompressed portion of S , which is a phrase. For $i = 1, 2, \dots, |P^0|$, one can find P^i , the longest prefix of $S^U[i + 1 : |S^U|] = P^i$ which is a phrase, iteratively as follows. Start from the node r that was reached in the previous iteration (it should represent P^{i-1} or some earlier P^j). Contract by exactly one character through the reverse trie and extend by as many characters as possible through the trie to reach a new node q . If an extension is not possible, assign $q = r$ and make a note through which P^j the node r was reached. Iterate.

The prefix P^j which is noted in the final iteration (at which $i = |P^0|$) is the phrase to be parsed by the one-step lookahead greedy parsing.

6.4.4 Karp–Rabin Fingerprints

Karp and Rabin [13] introduced fingerprints for simple and efficient string search via hashing, enabling one to perform both expand and contract operations as per the trie–reverse trie pair.

In the context of dictionary-based compression one can employ a hash table H of size p for a sufficiently large random prime number p to store all phrases in D . This is done by treating each phrase P as a number in base $|\Sigma|$ and using the hash function $h(P) = P \pmod p$. Clearly inserting and deleting a phrase P in the hash table takes optimal $O(|P|)$ time. It is also possible to implement contract and expand operations in $O(1)$ expected time each [29] in the spirit of the

trie-reverse trie pair as follows. Given string Q it is possible to compute the hash value of string Qa , the concatenation of Q with an arbitrary character a , from the hash value $h(Q)$ in $O(1)$ time by using the fact that $h(Qa) = [h(Q) \cdot |\Sigma| + a] \pmod{p}$. Similarly one can compute the hash value of string $Q[2 : |Q|]$ in $O(1)$ time via the fact that $h(Q[2 : |Q|]) = [h(Q) - |Q| \cdot |\Sigma| \cdot a] \pmod{p}$.

6.5 BENCHMARK PROGRAMS AND STANDARDS

Many variants of the LZ77 and LZ78 methods have become benchmarks for dictionary-based compression. Almost all the archiving programs such as `zip`, `PKZip`, `LHarc`, `ARJ`, and `gzip` make use of the LZ77 algorithm (or its LZSS variant) after the files are merged together in a reversible fashion. Below we describe some of the better known benchmark compression programs and some compression standards based on the dictionary compression paradigm.

6.5.1 The `gzip` Program

The `gzip` compression program was distributed by the Free Software Foundation as a utility for the GNU operating system. It provides a modification of the LZSS compression method, often giving good compression results in a reasonable amount of time. The main innovation in `gzip` lies in searching for the “best” match in the uncompressed portion of the text. For this purpose `gzip` builds a hash table to store three-character phrases observed in the compressed part of the input. The hash table stores (the location of) every occurrence of each three-character phrase in a linked list where the order of the occurrences is determined in a move-to-front fashion (i.e., the more recently an occurrence has been accessed, the closer to the top of the link list it is located). This approach improves the search time for cases where more recently accessed three-character phrases (and characters following them in the input string) have a higher chance of being accessed again. It is possible to limit the length of the linked lists corresponding to each hash value so as to put an upper bound on the time to perform a search with a given three-character phrase. This, however, may result in a decrease in compression performance.

The `gzip` program is also capable of performing greedy parsing with one-step lookahead and return not the longest phrase but one which is shorter if the next phrase obtained is composed of a single character. Note that for the general implementation of LZ77, such a lookahead cannot provide any improvement in compression; this becomes an issue in `gzip` only because it does not maintain all the dictionary entries (i.e., all substrings of the compressed portion of the input) implicitly suggested by LZ77. The `gzip` program further tries to improve the compression performance of LZ77 by Huffman coding the *offset* and *length* entries of each codeword separately.

6.5.2 The `compress` Program

Developed as a UNIX operating system utility, the `compress` program is essentially based on the LZW compression method. The dictionary is initialized with 256 entries corresponding to the possible combinations of 8 bits (that is, that this program processes the file to be compressed in a byte-wise fashion). The maximum size of the dictionary is variable, so it could be set to a value in the range $2^9, 2^{10}, \dots, 2^{16}$. At any step of the execution of the algorithm each codeword is encoded by $b = \lceil \log_2(|D| + 1) \rceil$ bits, where $|D|$ is the size of the dictionary. Once the dictionary size reaches the upper limit, a statistical check is performed at regular intervals so as to determine whether reasonable compression is being achieved without any alterations in the existing state of the dictionary. If the compression ratio falls below a preset threshold, the dictionary is reset to the initial state.

6.5.3 The **GIF** Image Compression Standard

Although it was developed more than a decade ago by CompuServe, the **GIF** compression standard still remains the most widely accepted lossless image compression standard and is used extensively in many applications and on the Internet. Based mainly on the LZW compression method, it employs the compress-like variable bit encoding of codewords while enforcing a maximum dictionary size of 4096. The dictionary is initialized with 2^r entries corresponding to all possible combinations of r bits, where r is the minimum number of bits required to represent any pixel in the image. The next two entries contain the *clear code* and the *end of image* codewords. Thus the number of bits b required to encode a codeword is set initially to $r + 1$. As the compression proceeds and new entries are continuously added to the dictionary, the value of b is recomputed to ensure that no more than sufficient bits are being used to encode the codewords. The dictionary remains static once it becomes full unless a clear code is sent to the decoder to signal the resetting of the dictionary and the initialization of all the relevant parameters. This compression method is particularly useful in lossless compression of synthetic (i.e., computer-generated) images which consist mainly of smooth regions of same color intensity values. Since the image is scanned in a horizontal direction such that two vertically oriented neighboring pixels in an image are set apart by the number of columns of the image, this compression scheme exploits only horizontal redundancies and not the vertical redundancies.

6.5.4 Modem Compression Standards: **v. 42bis** and **v. 44**

It is a well-known fact that the effective speed of signal transmission through a modem depends largely on the data compression scheme used therein. Compression methods based on dictionaries have been at the heart of two modem compression standards: the older and widely accepted **v. 42bis** and the newer **v. 44**. The telecommunications standardization division of the International Telecommunication Union (formerly CCITT) has approved a recommendation **v. 44** for data compression procedures over telephone lines recently (in November 2000). Where the **v. 42bis** standard was based on the LZW compression scheme patented by the Unisys Corp., the new **v. 44** recommendation is based on the lesser known Lempel–Ziv–Jeff–Heath (LZH) compression method patented by the Hughes Network Systems. The LZH method can basically be regarded as an LZ78 variant with the provision of extending a phrase by its first few symbols. The **v. 44** standard appears to be applicable to a variety of communication applications and is claimed to be particularly efficient for typical Internet downloads such as HTML files and images.

6.6 REFERENCES

1. Bell, T. C., 1986. Better opm/l text compression. *IEEE Transactions on Communications*, Vol. COM-34, pp. 1176–1182.
2. Brent, R. P., 1987. A linear algorithm for data compression. *Australian Computing Journal*, Vol. 19, pp. 64–68.
3. Cohn, M., and R. Khazan, 1996. Parsing with suffix and prefix dictionaries. In *IEEE Data Compression Conference*.
4. Chen, X., S. Kwong, and M. Li, 2000. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB-00)*, April 8–11 2000. (R. Shamir, S. Miyano, S. Istrail, P. Pevzner, and M. Waterman, Eds.), p. 107, ACM Press, New York.
5. Farach, M., 1997. Alphabet independent suffix tree construction. In *IEEE Symposium on Foundations of Computer Science*.

6. Fiala, E. R., and D. H. Greene, 1989. Data compression with finite windows. *Communications of the ACM*, Vol. 32, pp. 490–505.
7. Grumbach, S., and F. Tahi, 1994. A new challenge for compression algorithms: Genetic sequences. *Inf. Proc. and Management*, Vol. 30, No. 6, pp. 875–886.
8. Hariharan, R., 1997. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, Vol. 55, No. 1, pp. 44–69, August 1997.
9. Horspool, R. N., 1991. Improving lzw. In *IEEE Data Compression Conference*.
10. Horspool, R. N., 1995. The effect of non-greedy parsing in Ziv–Lempel compression methods. In *IEEE Data Compression Conference*.
11. Hartman, A., and M. Rodeh, 1985. Optimal parsing of strings. *Combinatorial Algorithms on Words*, pp. 155–167.
12. Jacquet, P., and W. Szpankowski, 1995. Asymptotic behavior of the Lempel–Ziv parsing scheme and digital search trees. *Theoretical Computer Science*, Vol. 144, pp. 161–197.
13. Karp, R., and M. O. Rabin, 1987. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development*, Vol. 31, No. 2, pp. 249–260.
14. Louchard, G., and W. Szpankowski, 1995. Average profile and limiting distribution for a phrase size in the Lempel–Ziv parsing algorithm. *IEEE Transactions on Information Theory*, Vol. 41, No. 2, pp. 478–488, March 1995.
15. McIlroy, M. D., and J. L. Bentley, 1999. Data compression using long common strings. In *IEEE Data Compression Conference*.
16. McCreight, E. M., 1976. A space economical suffix tree construction algorithm. *Journal of the ACM*, Vol. 23, No. 2, pp. 262–272, April 1976.
17. Matias, Y., and S. C. Sahinalp, 1999. On optimality of parsing in dynamic dictionary based data compression. In *ACM-SIAM Symposium on Discrete Algorithms*.
18. Miller, V. S., and M. N. Wegman, 1985. Variations on a theme by Lempel and Ziv. *Combinatorial Algorithms on Words*, pp. 131–140.
19. Rodeh, M., V. Pratt, and S. Even, 1981. Linear algorithm for data compression via string matching. *Journal of the ACM*, Vol. 28, No. 1, pp. 16–24, January 1981.
20. Savari, S., 1997. Redundancy of the Lempel–Ziv incremental parsing rule. In *IEEE Data Compression Conference*.
21. Storer, J. A., and T. G. Szymanski, 1982. Data compression via textual substitution. *Journal of the ACM*, Vol. 29, No. 4, pp. 928–951, October 1982.
22. Storer, J. A., 1998. *Data Compression: Methods and Theory*. Computer Science Press, 1988.
23. Sahinalp, S. C., and U. Vishkin, 1994. Symmetry breaking for suffix tree construction. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing: Montreal, Quebec, Canada, May 23–25, 1994*, pp. 300–309, ACM Press, New York.
24. Ukkonen, E. 1995. On-line construction of suffix-trees. *Algorithmica*, Vol. 14, No. 3, pp. 249–260. [TR A-1993-1, Department of Computer Science, University of Helsinki, Finland]
25. Weiner, P., 1973. Linear pattern matching algorithms. In *IEEE Symposium on Switching and Automata Theory*, pp. 1–11.
26. Welch, T. A., 1984. A technique for high-performance data compression. *IEEE Computer*, pp. 8–19, January 1984.
27. Welch, T. A., 1985. U.S. Patent 4,558,302, December 1985.
28. White, H. E., 1967. Printed English compression by dictionary encoding. *Proceedings of the IEEE*, Vol. 55, pp. 390–396.
29. Sahinalp, S. C., Y. Matias, and N. Rajpoot, 2001. The effect of flexible parsing for dynamic dictionary based data compression. *ACM Journal of Experimental Algorithms*.
30. Ziv, J., and A. Lempel, 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. IT-23 No. 3, pp. 337–343, May 1977.
31. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. IT-24 No. 5, pp. 530–536, September 1978.

This Page Intentionally Left Blank

Burrows–Wheeler Compression¹

PETER FENWICK

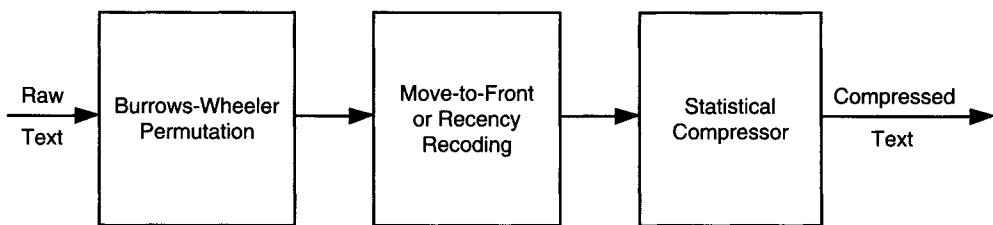
7.1 INTRODUCTION

Block-sorting compression, or “Burrows–Wheeler compression” is a relatively new algorithm of good compression and speed, first presented by Burrows and Wheeler in 1994 [1], although Wheeler had discovered the basic algorithm some 10 years earlier. In contrast to most other compression algorithms it treats the incoming text as a block, or sequence of blocks, with transformations on each block.

An initial study of Burrows–Wheeler compression was conducted by Fenwick, with a series of reports [2–4] and papers [5, 6]. Later work on the topic, by many other authors, will be cited throughout this chapter. Particular mention should be made of Seward [7], who took Fenwick’s initial implementations and produced a commercial-strength implementation, BZIP, released through the Free Software Foundation.

Most of the compression results here will be for the Calgary Corpus [8], a long-standing and somewhat arbitrary collection of files which has become a *de facto* standard for compression evaluation. Some results also use the newer Canterbury Corpus [9], a rather more systematically designed collection of files, chosen to reflect current compression needs and practices. Compression will be stated as bpc (bits per character), which is generally equivalent to the older term “bits per byte.”

¹ Author’s note: The material of this chapter, while quoting extensively from other work, is in part a summary of my own experience and thoughts in working with the Burrows–Wheeler compression algorithm. Some of it is accordingly rather less formal in style than might otherwise be the case, as I give more personal opinions on various aspects. Where my own work already appears in the public domain it is cited in the normal way, but unpublished material simply refers to “the author.”

**FIGURE 7.1**

Stages of Burrows–Wheeler or block sorting compression.

7.2 THE BURROWS–WHEELER ALGORITHM

In its original form, the compression algorithm processes the text in the three stages shown in Fig. 7.1. Later developments may vary from this basic scheme.

For now we just outline each of these three steps:

1. The heart of the compression algorithm is the Burrows–Wheeler transformation on the incoming text, which permutes or reorders the text into a form which is especially suited to processing by the following two stages, with clusters of similar symbols and many symbol runs.
2. The permutation step is followed by a recoding of the permuted text by a Move-To-Front (MTF) or recency recoder. (For a given symbol x , if n different symbols have been seen since the last occurrence of x , then the symbol x is recoded to the integer n .) The essence of this step is that recent symbols recode into small integers and symbol runs recode into runs of zeros; the MTF output is dominated by small values and especially zeros.
3. The final step is some form of statistical compressor. An adaptive Huffman or similar bit-oriented coder is appropriate where speed is more important, but an arithmetic coder may be better if good compression is emphasized.
4. Although it is not shown in the diagram, many implementations use run-length encoding in one or more parts of the process. For example, many statistical encoders are assisted by transforming the zero-runs from the MTF stage. Again, run-encoding the initial text may greatly accelerate some implementations of the permutation stage.

7.3 THE BURROWS–WHEELER TRANSFORM

Burrows–Wheeler, or block-sorting, compression is based upon two quite separate transformations, a forward transformation, which permutes the input data into a form which is easily compressed, and a matching reverse transformation, which recovers the original input from the permuted data. The data is always considered in blocks, which may be as large as the entire file or may be a few tens or hundreds of kilobytes. (The discussion here is based on Fenwick’s reports [2–4].)

7.3.1 The Burrows–Wheeler Forward Transformation

There are two approaches to describing the forward transformation. Burrows and Wheeler describe it as the steps:

1. Write the input as the first row of a matrix, one symbol per column.
2. Form all cyclic permutations of that row and write as them as the other rows of the matrix.
3. Sort the matrix rows according to the lexicographical order of the elements of the rows.
4. Take as output the final column of the sorted matrix, together with the number of the row which corresponds to the original input.

In terms which are more familiar to workers in data compression, the transformation may be described as:

1. Sort the input symbols, using as a sort key for each symbol the symbols which immediately follow it, to whatever length is needed to resolve the comparison. The symbols are therefore sorted according to their *following* contexts, whereas conventional data compression uses the *preceding* contexts. (Some implementations do use preceding contexts, but the discussion is easier with following contexts.)
2. Take as output the sorted symbols, together with the position in that output of the last symbol of the input data.

In either case the effect of the sorting is to collect together similar contexts. The (assumed) Markov structure of the input implies that only a few symbols are likely to occur in association with adjacent contexts. Any region of the permuted file will probably contain only a very few symbols and this locality can be captured with a Move-to-Front compressor.

7.3.2 The Burrows–Wheeler Reverse Transformation

Perhaps the most surprising thing about the forward transformation described above is that it is easily reversed! The reverse transformation depends on two observations:

1. The transformed input data is a permutation of the original input symbols.
2. Sorting the permuted data gives the first symbol of each of the sorted contexts.

But the transmitted data is ordered according to the contexts, so the n th symbol transmitted corresponds to the n th ordered context, of which we know the first symbol. So, given a symbol s in position i of the transmitted text, we find that position i within the ordered contexts contains the j th occurrence of symbol t ; this is the next emitted symbol. We then go to the j th occurrence of t in the transmitted data and obtain its corresponding context symbol as the next symbol. The position of the symbol corresponding to the first context is needed to locate the last symbol of the output and from there we can traverse the entire transmitted data to recover the original text.

7.3.3 Illustration of the Transformations

To illustrate the operations of coding and decoding we consider the text “mississippi”, as illustrated in Fig. 7.2. The first context is “imississip” for symbol “p”, the second is “ippimissis” for symbol “s”, and so on. The permuted text is then “pssmipissii”, and the initial index is 5 (marked with “ \rightarrow ”), because the fifth context corresponds to the original text.

To decode we take the encoded string “pssmipissii”, sort it to build the contexts (“iiimppssss”), and then build the links shown in the last column in Fig. 7.2. The four “i” contexts link to the four “i” input symbols in order (to 5, 7, 10, and 11, respectively). The “m” context links to the only “m” symbol, and the two “p”s and four “s”s link to their partners in order.

Symbol	Context	Index	Symbol	Context	Link
p	imississip	1	p	i...	5
s	ippimmissis	2	s	i...	7
s	issippimis	3	s	i...	10
m	ississippi	4	m	i...	11
→i	mississipp	5	i	m...	4
p	pimississi	6	p	p...	1
i	ppimississ	7	i	p...	6
s	sippimissi	8	s	s...	2
s	sissippi	9	s	s...	3
i	ssippimiss	10	i	s...	8
i	ssissippi	11	i	s...	9

FIGURE 7.2

The forward and reverse transformations.

To finally recover the text, we start at the indicated position (5) and immediately link to 4. The sorted received symbol there yields the desired symbol “m” and its immediately following context symbol, the fourth “i”. We then link to 11 get the “i”, and so on for the rest of the data, stopping on a symbol count or the return to the start of the file.

7.3.4 Algorithms for the Reverse Transformation

The forward transformation is largely concerned with sorting and will be described later. Here we describe the steps of the reverse transformation, assuming the notation:

- N denotes symbols in the file
- n denotes the symbols in the alphabet
- S denotes the array of received symbols S[1..N]
- T is the array of recovered symbols T[1..N]
- K denotes the counts of each symbol K[0..n-1]
- L denotes the links to resolve permutation L[1..N]
- M is the mapping array for symbols M[0..n-1]

The received (permuted) symbols are assumed in S, with counters, etc., appropriately initialized. The first step is to just count the occurrences of each symbol; it may be performed as the file is read in, decompressed, and the symbols recovered from the inverse Move-To-Front.

```
for(i = 1; i <= N; i++)
    K[S[i]]++; /* count input symbols */
```

The next step involves building the table of the initial positions of each context. Remember that these are ordered according to the lexicographical ordering of the symbols and that there are as many contexts starting with “a” as there are occurrences of “a” in the input.

```
M[0] = 1;
for (j = 1; j < n; j++)
    M[j] = M[j-1]+K[j-1];
```

We are now in a position to build the links which will be used to traverse the received data. The mapping array M initially points to the first positions of the contexts. As a symbol s is used from a context, M[s] is incremented so that it tracks the next context starting with “s”.

```
for(i = 1; i <= N; i++)
{
    s = S[i];           /* get current symbol */
    L[i] = M[s];        /* set link from mapping table */
    M[s]++;             /* increment mapping */
}
```

Although we never actually prepare an explicit list of the contexts (or rather their initial symbols), the steps so far are equivalent to using such a sorted list. Finally we can recover the original text.

```
ix = initial_position;          /* start at correct position */
for (i = 1; i <= N; i++)
{
    ix = L[ix];                /* step on via the links array */
    T[i] = S[ix];              /* copy symbol to output */
    if (ix == initial_position)
        break;                  /* an alternative termination */
}
```

There are three passes through the entire file (one of which may be combined with reading) and one pass through the alphabet. The operations are all quite simple and usually add little to the input/output costs of handling the file or, in most cases, to the cost of decompressing the file.

7.4 BASIC IMPLEMENTATIONS

We now describe an implementation of the entire compressor, generally leading to the version described by Fenwick [4–6]. It will bring in some of the experience gained from those early implementations and that acted a basis for later developments described in Section 7.5. When this version was completed in 1995 (published in 1996) it was nearly as good as the best PPM compressors at the time and certainly illustrated the potential of the Burrows–Wheeler or block-sorting algorithm.

7.4.1 The Burrows–Wheeler Transform or Permutation

In this section we deal only with the forward transform; the reverse transform is quite straightforward and independent of the forward implementation. In the original report Burrows and Wheeler present the algorithm in terms of sorting, but also mention that suffix trees provide an alternative and possibly better implementation. Much of the later work on the algorithm has actually dealt with suffix tree implementations. The author, however, with a greater interest in improving the final coding, has persisted with the sorting algorithm as described here.

All of the author’s implementations have used a combination of a bucket sort (65,536 buckets, based on symbol digrams) and the standard C *qsort* implementation of QuickSort. An initial pass over the input string builds up, for each digram, a table of the indices of occurrences of

Comparisons for string "mississippi"	Symbol	Word
	m	miss
	i	issi
First comparison →	s	ssis
	s	siss
	i	issi
	s	ssip
Next comparison →	s	sipp
	i	ippi

FIGURE 7.3

Example of character striping.

that diagram. The sort is a tag sort, permuting these indices. We also build a word array, striping symbols across successive words allowing 4 (or possibly 8) symbols to be compared at a time, with the comparison striding 4 or 8 words through the array, as shown in Fig. 7.3.

When presented with the string "mississippi", based on the digram "mi", the comparison procedure supplied to *qsort* will first test the word "ssis" (the first two letters must match because they are in the same digram bucket) and then the word "sipp". Only if these two both succeed will the main comparison loop be entered.

A walk through the digrams in order and the sorted tags within the digrams gives the complete ordering by context; the symbols immediately before each context are those emitted.

Two problems occurred with this sort routine. The first is that some implementations of *qsort* will compare an entity against itself; self-comparisons must be trapped at the very beginning of each string comparison (are the initial indices identical?).

The second problem is that some files, such as the Calgary Corpus *pic*, have long runs of identical symbols, and sorts on these runs may require enormous numbers of comparisons to resolve. In Fenwick's original implementation a run of four or six identical symbols always signals a run. All immediately following occurrences of that symbol are replaced by a run-length or symbol count (which may be zero). This preprocessing step is intended only to help files such as *pic* and has little effect on other files.

Wheeler [10] described a sorting technique which starts as a radix-256 sort, based on the initial symbols of the contexts and with symbols striped across successive long words, as described above. He then sorts the buckets into increasing size and sorts the smaller ones first. After each bucket is sorted, each of the words (32 bits) referred to by that bucket has its rightmost 24 bits replaced by its ordinal position in the sorted bucket, leaving the symbol itself in the leftmost 8 bits. Thus each symbol in the bucket is uniquely tagged according to its lexicographical ordering and whenever two already processed symbols are compared (as words of four symbols or as position+symbol) the comparison is resolved immediately.

7.4.2 Move-To-Front Recoding

The aim and object of any statistical text compressor is to produce a highly skewed symbol distribution so that some symbols are known to be much more likely than others in a particular context or position in the output. In general, the greater the skewness, the better the compression, as the few likely symbols can be encoded in very few bits. In the standard Burrows-Wheeler compressor this skewed distribution is produced by a MTF recoding of the permuted text.

Table 7.1 Move-To-Front Statistics from Calgary Corpus

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progI	progP	trans
Average MTF movement													
2.50	2.45	2.25	36.05	3.80	23.47	10.26	3.27	2.81	1.30	3.90	1.99	2.18	1.96
Fraction of MTF codes which are zero													
66.8%	49.8%	60.8%	35.8%	57.9%	50.6%	68.1%	58.4%	55.4%	87.4%	60.3%	72.9%	74.0%	79.2%

Symbol	MTF List Before	MTF List After	Emitted Code
p	abcd..	pabc..	?
s	pabc..	spab..	?
s	spab..	spab..	0
m	spab..	mspa..	?
i	mspa..	imsp..	?
p	imsp..	pims..	3
i	pims..	ipms..	1
s	ipms..	sipm..	3
s	sipm..	sipm..	0
i	sipm..	ispn..	1
i	ispn..	ispn..	0

FIGURE 7.4

Example of MTF recoding.

While the term Move-To-Front is strictly a descriptive one based on the usual implementation, the technically more accurate description is “recency” recoding. Each symbol is replaced by the number of different symbols which have appeared since its last occurrence in the data stream.

In Fig. 7.4 we show the symbols from the earlier Burrows–Wheeler permutation of “mississippi” and the recoding using a MTF list. (Visual inspection shows that it implements recency recoding.) The MTF list is initially some permutation of the full symbol alphabet. Each symbol is recoded as its current position in the MTF list; the list is then shuffled to bring that symbol to the front of the list, retaining the order of the others. Here the *first* occurrence of each symbol is coded as some arbitrary value, shown as “?” Thereafter, recent symbols tend to be coded into small values, irrespective and independent of the value for the first reference.

Move-To-Front recoding was first used in data compression by Bentley *et al.* [11], who used a word-based compressor with quite large movements of large objects through the MTF list or table. Much of their discussion concerned the merits of various data structures to maintain the table and allow updates at low cost. Here, however, the objects to be moved are simple (just single 8-bit bytes or symbols) and the average movement is small, often only two or three positions, as shown for the Calgary Corpus in Table 7.1, although the movement is much greater for the three less compressible binary files. The table also shows the proportion of the MTF codes which are zero; the overall average is about 62%.

The initial implementation used two parallel byte arrays: one the current list itself and the other a mapping array to find the current position of each byte or symbol value. Later implementations dispensed completely with the mapping array, with a simple search along the list to find the index

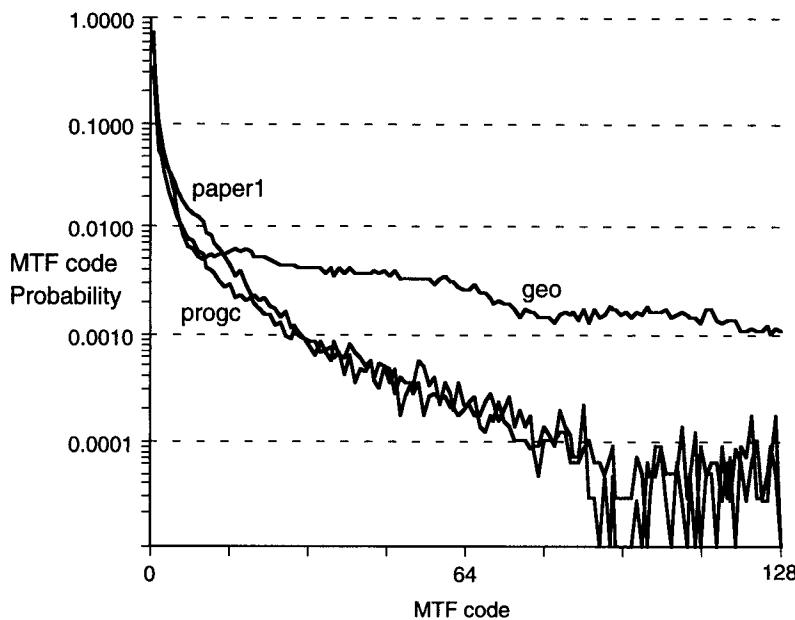


FIGURE 7.5
MTF code probabilities for three files.

of the symbol. The average overhead of the search is less than the average overhead of shuffling the mapping array.

7.4.3 Statistical Coding

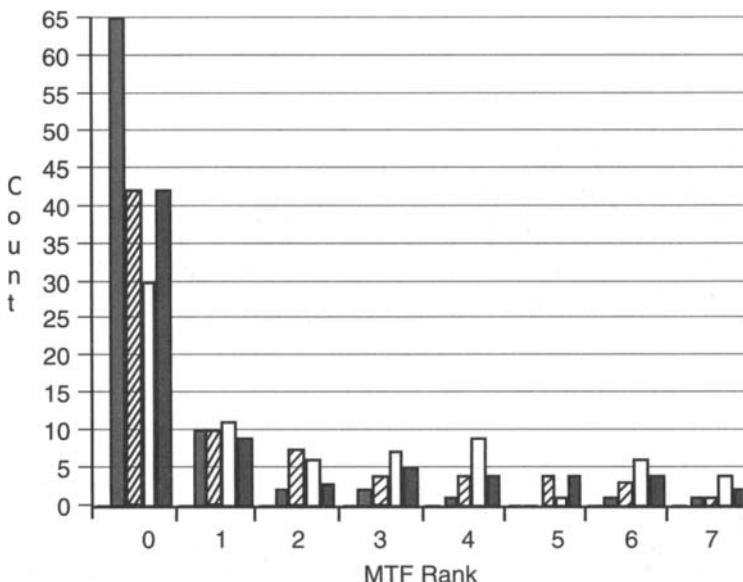
Most of the work on high-performance Burrows–Wheeler compressors has used arithmetic coders, or a complex of such coders, in the final stage. Huffman and similar bit-wise coders can be used, and are used, but will not be discussed here.

As stated earlier, good compression implies a skewed probability distribution of the encoded symbols. Normal text follows a Zipf distribution in which the r th most frequent symbol has a frequency proportional to r^{-1} . Work by Fenwick [4, 6] showed that the MTF output in Burrows–Wheeler compression has a distribution similar to r^{-2} , with the exponent being larger for more compressible files. The more skewed distribution and much greater range of probabilities have major consequences for a final arithmetic compressor.

Figure 7.5 shows the distribution of MTF code probabilities for three files of the Calgary Corpus. Two are text files (but the line for paper1 is generally above that for progc, showing its lesser compressibility) while geo has many symbols at higher ranks and is correspondingly less compressible. The text files show a range of probabilities exceeding 10,000:1.

Initial work (by the author) used a combination of Witten’s “CACM” arithmetic coder [12], in conjunction with a new data structure for holding the frequency counts[13] which allowed greater choice of the coding range and per-symbol increment. When Moffat *et al.* [14] released their improved coder, that was substituted and immediately gave much poorer compression! Solving this problem led to a much better understanding of Burrows–Wheeler compression.

Moffat’s coder is designed for a conventional PPM-style compressor where each context has its own coding model and statistics accumulate for many models as compression proceeds. Most versions of Burrows–Wheeler compression maintain just a single coding model (if only because it can be very difficult to decide when to switch between models, but see later work in Section 7.11).

**FIGURE 7.6**

MTF code frequencies for successive areas of a sorted file.

The coding statistics, however, are far from stable; as an entropy source the MTF output is far from ergodic. We find that a file, almost any file, transforms into bursts of high entropy on a background of quite low entropy. The difference between files is that in a relatively compressible file the bursts of high entropy tend to be shorter rather than lower in absolute value. The Moffat coder had to be redesigned to increment and scale counts in a manner similar to that of the original CACM coder.

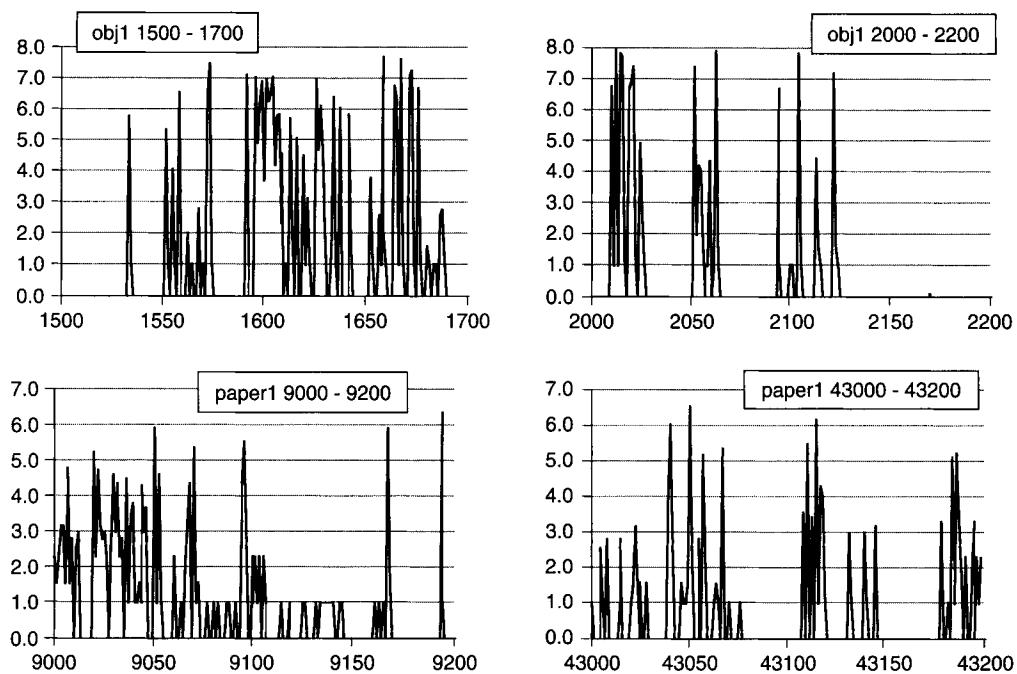
Figure 7.6 shows the counts of various small MTF ranks in four successive 100-symbol blocks of the file *paper1*. The variation is considerable, with many MTF counts varying by more than 2:1 over even this small section of file. Thus statistics accumulated for one section of the file may have to be adjusted, or even forgotten completely, quite soon afterward. The statistical coder needs considerable agility to track these changes.

Figure 7.7 shows the approximate coding cost in bits for each symbol of sequences of 200 symbols from the files *paper1* and *obj1*. We see a general mixture of extreme activity, alternating with very quiet periods, even for a relatively incompressible file like *obj1*. The horizontal axis is in each case the symbol position in the permuted file.

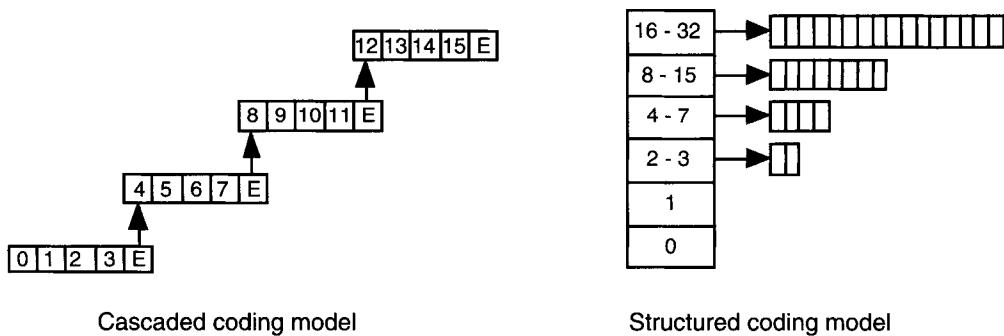
An arithmetic coding model for Burrows–Wheeler compression must combine the two attributes:

1. It must handle a wide range of symbol probabilities, from 10,000:1 for a typical text file to 65,000:1 for some binaries. This requires a significant difference between the limit for the accumulated count and the per-symbol increment. The ratio need not be as high as 10,000 or 50,000 to 1, but certainly should be high.
2. It must be able to adapt quickly as the output moves between the bursts and the quiescent background. Fast adaptation usually implies a small ratio of limit: increment to force frequent overflows and scaling of the counts.

At first sight the two requirements of range and agility are largely incompatible, but may be reconciled by combining coding models. Two forms are the cascaded model and the structured model, both outlined in Fig. 7.8. The extension to handle larger alphabets should be obvious.

**FIGURE 7.7**

Coding cost (bits per symbol) for sections of files obj1 and paper1.

**FIGURE 7.8**

Cascaded and structured coding models.

In both cases the barred rectangles are arithmetic coding models handling an appropriate number of symbols.

1. In the cascaded model each component handles a small range of consecutive values, with an “escape” to overflow into the next model. In the example here, coding a “13” would result in the codes E, E, E, 1 as the coder drives through the successive components of the whole model. If symbols in the range 0–3 are emitted, only the very first coding model is used and later models remain untouched. A sequence of larger symbols will enter the later coding models but the early models affect only the probability of the escape symbol(s). Successive levels of the cascade need not have the same size as in the example. It is probably more sensible to have them growing as do the leaf nodes of the structured model.

Table 7.2 Typical Parameters for Structured Coding Model

Group	Range	Count Limit	Count Increment
Trunk	—	2000	20
0	0	—	—
1	1	—	—
2	2–3	256	1
3	4–7	256	1
4	8–15	512	1
5	16–31	1024	1
6	32–63	2048	1
7	64–127	4096	1
8	128–255	8192	1

2. The structured model has a “trunk” model with several “leaf” models. Instead of a sequence of possible escapes, most symbols have an initial trunk code, followed by an appropriate leaf code coding a residue. For example “6” would be encoded as an initial “3” (element 3 of the trunk) and then “2” as the residue within the leaf. Exactly as with the cascaded model a sequence of small codes leaves the “high-value” portion of the model untouched. The structured model was first devised by Fenwick [6] and designed to equalize the probabilities served by each entry of the trunk model. In retrospect this equalization is relatively unimportant. Much more significant is the ability of the trunk model to decouple the effects of high and low code values.

Both of the models rely on one or more small coding models which handle only a few symbols and can adapt very quickly to changing statistics. Experience shows that both give a similar performance. Fenwick [3] used a cascade model with initial binary models feeding into a full background model, for a Calgary Corpus result of 2.363 bpc, which is very close to his final 2.338 bpc for the structured model. (Much of the difference probably comes from using run-encoding in the structured model but not with the cascade.)

By way of illustration, Table 7.2 shows typical arithmetic coding parameters for a structured model; the values for the leaf nodes are not critical.

7.4.3.1 Run-Length Coding

A final way of improving compression is to run-encode the streams of 0’s which dominate the MTF output. Strictly this should be unnecessary with a good arithmetic compressor, because over a stream of N consecutive 0’s, the cost per encoded digit tends to zero, while the cost of encoding any other non-0 symbol to terminate the run tends to $\log_2 N$ bits. But as the count of N can also be encoded in $\log_2 N$ bits, we can expect the costs to be similar. In practice the arithmetic coder does not adjust immediately to the ideal coding and an explicit run-length coding gives slightly better results.

A suitable method of encoding zero runs, suggested by Wheeler, is to expand the alphabet by one symbol so that most values x are encoded as the value $x + 1$. For a run of N zeros, the actual code values {0, 1} are used to encode the binary representation of $(N + 1)$, least significant bit first and omitting the most significant 1 bit. The leading 1 is implied by the code >1, which follows the run.

An example of this encoding is shown in Table 7.3. At the small price of increasing the alphabet by one symbol it gives a very efficient coding of runs of zeros, never increasing the data length. In Table 7.3 it saves 4 symbols out of the original 15. Using the run-length coding gave about a 1% improvement in compression.

Table 7.3 Wheeler's Run-Length Recoding of a Data Stream

Original Data	6	2	1	0	1	0	0	2	3	0	0	0	0	0	4
Recoded Data	7	3	2	0	2		1	3	4		0	1	5		

Table 7.4 Fenwick's 1996 Compression of Calgary Corpus Files: Average = 2.338 bpc

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	prog1	prog2	trans
1.946	2.391	2.030	4.500	2.499	3.865	2.457	2.458	2.413	0.767	2.495	1.715	1.702	1.495

One final, small, improvement comes from limiting the alphabet to 128 symbols for text files, giving the final results for Fenwick's 1996 implementation [4–6] as shown in Table 7.4.

7.5 RELATION TO OTHER COMPRESSION ALGORITHMS

Almost as soon as it was first described, Cleary and Teahan [15] showed that the Burrows–Wheeler transform had a formal correspondence to their PPM* compressor and was described by a similar context tree. However, an important difference lies in the manner by which that tree is used and traversed.

In on-line or sequential compressors such as PPM the tree is built “on the fly” and functions as a storage or repository for information on contexts which have been seen already. The active area moves all around the tree as compression proceeds, moving to an existing leaf or context for each symbol and extending that information or extending the tree in accordance with the current symbol and its context.

In Burrows–Wheeler compression, the tree is built, explicitly or implicitly, by the initial transform and is then traversed in lexical order. The context described by a given node may be extended into longer contexts for a while but once left is abandoned entirely and never revisited. The context tree of PPM is thereby replaced by a stack and the main function of the tree, storing context history and statistics for future reuse, is eliminated completely.

Effros [16] shows the equivalence of the context trees of PPM*, the suffix trees of some Burrows–Wheeler implementations, and the pattern-matching trees often used in Ziv–Lempel compressors, emphasizing the essential unity of these three apparently disparate methods. She then uses this knowledge to implement a compressor combining “PPM performance with BWT [Burrows–Wheeler Transform] complexity,” albeit closer to PPM than to BWT.

7.6 IMPROVEMENTS TO BURROWS–WHEELER COMPRESSION

We now take the compressor of the previous section and examine various improvements which have been proposed. The paper by Deorowicz [17] includes an excellent review of much of this material and readers should consult that paper for further details. In line with the compression stages as used above, the improvements are described here under the broad headings of:

1. Preprocessing the input.
2. The Burrows–Wheeler transform.

3. The Move-To-Front operation.
4. The final statistical compressor.

In fact the division is not as simple as this because often a problem in one area must be solved or have repercussions in another.

One point which must be raised is the question of overheads. It is all too easy to propose a solution which looks good until one counts the cost of the overheads. For example, suppose that we want to recode the input in some way and experiments show that this recoding improves compression of text files by 1%, a possibly useful amount. A text file has an alphabet of around 100 symbols, or perhaps a few less. The simplest way of describing the recoding is to transmit a recoding vector for these 100 symbols. This is unlikely to be very compressible and adds 100 characters to the compressed data. Given that ordinary text compresses at around 2.5 bpc, the 100 characters used to describe the recoding is equivalent to 320 characters of input text. Only if the input file exceeds 32,000 characters does the 1% gain exceed the 100-byte cost. Although some clever techniques may reduce the 100-byte description overhead, the overhead is still a cost which must be balanced against the gain. All too often the benefits are minimal.

7.7 PREPROCESSING

Two of the simpler types of preprocessing have been mentioned already, modifying the compression for a partial input alphabet and run-encoding the input before the permutation step. Partial input alphabets will be considered in Section 7.9 when the Move-To-Front operation is considered. Run-length encoding was introduced into the preprocessor only to aid in sorting files such as `pic`, which have long symbol runs. It was apparent from the beginning that run-length encoding destroys some of the symbol context and reduces compression by perhaps 0.1%. Balkenhol *et al.* [18] suggest that run-encoding should be avoided if it reduces the file size by less than 30%. This step is in any case unnecessary if a suffix-tree algorithm is used for the permutation.

More significant preprocessing was proposed by Chapin and Tate [19]. Their techniques apply some permutation (or Caesar cipher) to the input text with the idea of minimizing the costs of the MTF recoding. If symbols which are in some sense “similar” are brought together in the recoding, the cost of moving from one to the other in the MTF recoding will be smaller and the compression will improve. They found that just bringing the vowels together (a permutation to “aeioubcdfg . . .”) improved compression of text files by 0.3–0.5% but a full analysis, akin to a Traveling Salesman Problem, had less effect.

Overall, they improved the compression from 2.410 to 2.340 bpc for the Calgary Corpus, an improvement of about 3% in their implementations. Unfortunately all of the recoding methods must transmit the recoding information. This may be a simple codebook number (for a predefined recoding like `aeioubcdfg . . .`) but may require a dictionary containing the entire input alphabet. When we consider that Chapin reports that the saving of a recoding seldom exceeds 100 bytes for the smaller text files, it is doubtful whether anything more complex than a simple “vowel” codebook heuristic is worthwhile.

7.8 THE PERMUTATION

We might consider that an ideal permutation simply lists the input symbols or letters with their frequency counts, followed by the reordering rules. Unfortunately for most permutations the reordering rules are comparable in size to the original text! It appears that the Burrows–Wheeler

transform is almost unique; it is conceptually simple to perform and invert and adds just one integer of overhead to facilitate the inverse transform. Thus while we must remain open to the possibility of a better permutation algorithm, Burrows–Wheeler remains the best known.

There are nevertheless two other permutations to consider here

- Chapin and Tate [19] also introduce the idea of a “reflected sort,” analogous to the reflected binary or Gray code. In the Gray code, representations of successive integers differ in at most one place. Chapin modifies the sort comparison to alternate increasing and decreasing comparisons on successive bytes of the comparand strings. To quote Chapin “Whenever a symbol in a column changes between string i and string $i + 1$, the sort order of all following columns is inverted.”
- Arnavut and Magliveras [20] describe the *Lexical Permutation Sorting Algorithm*, a generalization of the Burrows–Wheeler transform.

Suffix trees were introduced by Burrows and Wheeler as an alternative to the explicit sorting algorithm. In one sense the suffix trees change nothing, being just an alternative way of implementing the Burrows–Wheeler transform. But particularly in the worst case they are faster and may be more space-efficient than a sorting implementation. They are therefore preferred by many authors for situations where compression speed is important.

Several authors consider other improvements to the sorting algorithm. Schindler [21] sorts using finite-length contexts as the keys, but at the cost of a more expensive reverse transform and slightly poorer compression. Sadakane [22] describes a fast and economical algorithm for sorting suffix arrays, suggesting that the *average match length* or average number of comparisons needed to verify the correct sorting is a measure of the difficulty of producing the Burrows–Wheeler transform. Seward [23] compares several sorting algorithms.

One final point to consider is the direction of the sort comparison (or its equivalent with suffix trees). One natural direction is a conventional lexicographical comparison in the forward direction (so that “somewhat” precedes “somewhere”). But equally natural is a reverse comparison (“anywhere” follows “somewhere”) because in compression we traditionally lead into the symbol under consideration. In his very early work on the entropy of English, Shannon [24] found that although the second approach is more natural for people to use, there is little real difference between the prediction ability of the two directions.

For the most part, the choice seems to be according to the whim of the implementer. Table 7.5 shows some results from Fenwick [3], for a very early version of the compressor and also for a much later version of the compressor with better final coding and other optimizations. Both cases show results for forward comparisons (trailing contexts) and reverse comparisons (leading contexts) of the incoming text.

Boldface text in Table 7.5 indicates where the compression is better by about 1% than in the other direction. In few cases are the differences significant, except in the file geo, which shows a change of nearly 4% between the two directions. This difference has been noted by

Table 7.5 Comparisons of Context Directions for Two Compressors

	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	prog1	prog2	trans	Average
Early compressor, little optimization															
Forward	2.132	2.524	2.198	4.810	2.678	4.202	2.709	2.606	2.570	0.830	2.680	1.854	1.837	1.613	2.517
Reverse	2.182	2.549	2.218	4.748	2.695	4.201	2.730	2.629	2.593	0.830	2.683	1.851	1.838	1.625	2.527
Later compressor, more optimization															
Forward	1.926	2.356	2.012	4.430	2.464	3.796	2.433	2.439	2.387	0.753	2.476	1.697	1.702	1.488	2.311
Reverse	1.953	2.363	2.012	4.268	2.476	3.765	2.478	2.464	2.420	0.758	2.478	1.697	1.716	1.503	2.311

C2	66	D8	00	C2	34	DC	00	42	10	60	00	42	35	4C	00
42	32	54	00	42	38	BC	00	42	63	9C	00	42	96	3C	00
42	99	28	00	42	9C	8C	00	42	9B	B8	00	42	89	BC	00
42	6C	D4	00	42	48	90	00	41	6B	40	00	C2	2D	90	00

FIGURE 7.9

Sixty-four consecutive bytes from the file geo, broken into floating point values.

several authors; for example, Balkenhol *et al.* [18] recommend using leading contexts or reverse comparisons where the file has an alphabet of 256 symbols or nearly so. However, obj1 and obj2 also have large alphabets and obj1 shows a negligible difference, while obj2 is better with trailing contexts or forward comparisons.

The explanation is just that geo consists of 4-byte (32 bit) floating-point numbers whose least significant (rightmost) byte is usually zero and independent of the next byte of that word. The zero byte is, however, adjacent to the first byte of the *next* word, which has relatively few values and is a good predictor of the zero byte *preceding* it in the file, as shown in Fig. 7.9. Working only from alphabet size is quite misleading. While some files do compress much better with comparisons in one direction, that direction must be decided individually, on a case by case basis. (In any case it should be noted that just reversing the context direction for geo typically gives an improvement of 0.012 bpc on the Calgary Corpus average, which is a significant part of the improvement quoted by many authors.)

7.8.1 Suffix Trees

In their initial paper Burrows and Wheeler suggested the use of suffix trees (see McCreight [25]) as an alternative to an explicit sort. That route has been followed by many workers, especially by those interested in improving the compression speed. But while they can give good speed, the suffix-tree algorithms often have high memory consumption. Larsson and Sadakane [26] have shown that it is often better to use the more complex algorithms which are non-linear in time. Some examples of algorithms quoted by Deorowicz are as follows:

- Manbers–Myer [28] has a worse-case time complexity $O(n \log n)$ and memory complexity $8n$.
- Bentley–Sedgwick [29] has a worst-case complexity $O(n^2)$, an average complexity $O(n \log n)$, and memory consumption $5n$ (plus a quicksort stack).
- Sadakane [22] describes an algorithm for producing a *suffix array*, as an alternative to a suffix tree. His measurements show a generation time which is generally the best or second best of the Manbers–Myer and Bentley–Sedgwick algorithms and an algorithm of Larsson [31].
- Larsson and Sadakane [26] describe an algorithm with worst-case complexity $O(n \log n)$ and memory complexity $8n$.

7.9 MOVE-TO-FRONT

The Move-To-Front step is usually regarded as an essential part of Burrows–Wheeler. That this is not so was demonstrated by Fenwick [3], who without much effort achieved a compression of 2.52 bpc on the Calgary Corpus in a Burrows–Wheeler compressor with no MTF stage.

Although not a good result by PPM and Burrows–Wheeler standards, it is nevertheless quite respectable by most compression standards.

Many authors have looked very suspiciously at the MTF recoding, realizing that by mixing the information from many different contexts it actually involves a considerable loss of possibly useful information. Despite this doubt, most have accepted the Move-To-Front recoding as necessary (a necessary evil?) and a great deal of work has been expended on improving the Move-To-Front step, with varying amounts of success. But more recently Wirth has applied PPM techniques and shown that excellent performance is possible with no MTF at all. We will defer that topic until later.

7.9.1 Move-To-Front Variants

Move-To-Front is one of the classical examples of a self-organizing list structure which dynamically restructures itself to facilitate access to recent or seemingly important data. It is the exact analog of the hierarchical caches found in modern computers, but providing a smooth transition instead of distinct levels. In comparison with many other self-organizing list structures (see Fenwick [32]), it is characterized by a very rapid movement of referent data to the preferred position at the front of the list. Many authors look with suspicion at this aspect and propose schemes with slower movement. Some examples developed for Burrows–Wheeler compression are as follows:

- **Move-One-From-Front (M1FF).** If the symbol σ is at rank 2 or higher, move it to position 1; otherwise move it to position 0 [18].
- **Move-One-From-Front-2 (M1FF2).** If σ is at rank 2 or higher and if the head symbol was requested last time or the time before, move it to position 1; otherwise move it to position 0.
- **Best x of $2x - 1$.** Put σ in front of all others that have been requested the minority of (fewer than x) times in the last $2x - 1$ requests of themselves and σ [33].
- **Best 2 of 3.** The previous method, with $x = 2$.

The author has developed another method described as “sticky MTF,” which works very well on most files. In this method the symbol is brought to the front immediately, but if it is not used on the very next symbol it is moved back to about 40% of its original position. The method is unpublished, although it was communicated privately to Seward and used in some of the BZIP compressors.

To justify sticky MTF, note that for most text files there is a 60% probability that a symbol, having been moved to the list head, will be reused immediately. It therefore makes sense to move it right to the front because this is where it will be needed. If, however, it is not reused immediately, the symbol will “pollute” the MTF list until it shuffles back behind the active symbols. Each one of these more recent symbols will incur a cost from the presence of this unwanted symbol. We therefore move the unused symbol well out of the way, far enough to clear the MTF “working set” but not so far as its original position, because it is likely to be reused. A return to about 40% of its original rank is a good compromise.

Results for both the Calgary Corpus and the newer Canterbury Corpus are shown in Table 7.6. With one glaring exception, sticky MTF gives an improvement of about 1% across most files. That exception is the file exc1 from the Canterbury Corpus, a file of spreadsheet data. Examination of the MTF output from this file shows that while most of the MTF is very quiet with most values predicted very well, it has large bursts of very high entropy, higher than is usual in other files. Many symbols are referred to once and then again after a break of a few other intervening symbols. With “non-sticky” MTF those symbols remain near the head of the list and

Table 7.6 Results from Sticky MTF on Calgary and Canterbury Corpora

Files from Calgary Corpus															
	bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progI	progP	trans	Average
Normal	1.943	2.390	2.037	4.482	2.497	3.865	2.456	2.452	2.411	0.779	2.492	1.708	1.699	1.491	2.336
Sticky	1.927	2.357	2.014	4.428	2.465	3.797	2.433	2.440	2.389	0.753	2.478	1.698	1.702	1.489	2.312
Reduction	0.8%	1.4%	1.1%	1.2%	1.3%	1.8%	0.9%	0.5%	0.9%	3.5%	0.6%	0.6%	-0.2%	0.1%	1.0%
Files from small Canterbury Corpus															
	csrc	excl	fax	html	lisp	man	play	poem	sprc	tech	text			Average	
Normal	2.081	0.971	0.779	2.436	2.536	3.108	2.508	2.390	2.608	1.993	2.249			2.151	
Sticky	2.097	1.237	0.753	2.410	2.547	3.098	2.483	2.361	2.571	1.969	2.228			2.159	
Reduction	-0.8%	-22%	3.5%	1.1%	-0.4%	0.3%	1.0%	1.2%	1.4%	1.2%	0.9%			-0.4%	

are accessed with low cost. With sticky MTF they are well removed from the head and are much more expensive to recover. This file is a salutary lesson in the ability of special cases to defeat optimization!

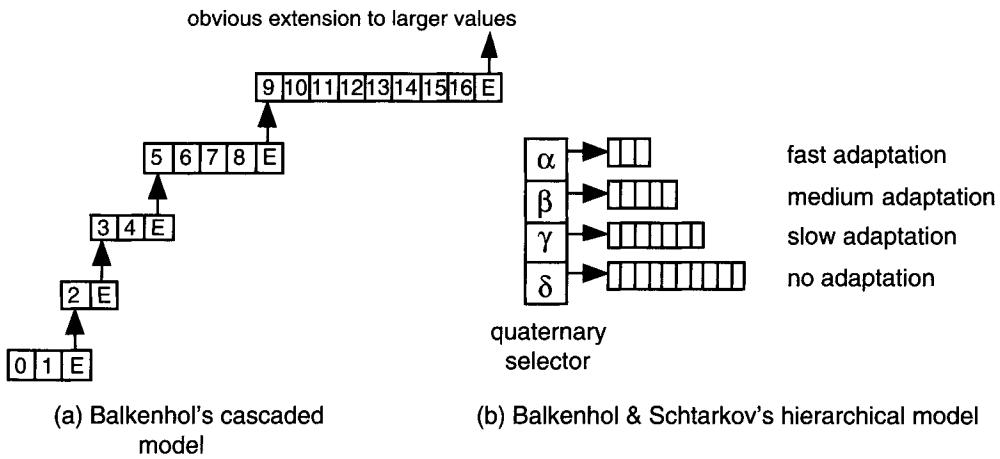
Another technique which impinges on MTF is to restrict the alphabet. This has a very small one-off effect for each symbol as it may have to move a shorter initial distance when it becomes active. (Most text files use only three or four of the ASCII “control” symbols and packing these reduces the initial MTF movement by about 30 positions.) Reducing the alphabet often has a useful effect on the final statistical compressor as it means that many symbols can be omitted entirely and do not clutter the model with available but unused counts.

Some caution is needed though in transmitting the alphabet of active symbols, as sending information on the reduced alphabet may cost more than any gain. The author has used a system which sends a bit vector of active symbols. Explicit full-mode (>240 symbols) and half-mode ($120 < n < 128$) cases are signaled if appropriate to avoid the cost of a bit vector. It gives an improvement of about 0.05 bpc on the Calgary Corpus.

7.10 STATISTICAL COMPRESSOR

Early work by Fenwick [2] indicated that the output of the Burrows–Wheeler permutation is compressed very poorly by “good” compressors such as PPM. Wirth [36] interprets this negative result as due to the context mechanism becoming overwhelmed. Although much of the input context structure is absorbed by the permutation, there is still some context structure in the recoded MTF output. The problem is to interpret that structure, while keeping the number of contexts within reasonable bounds.

A major work on the statistical compressor is by Balkenhol *et al.* [18]. They note that the output from the MTF recoding may be loosely divided into “good” and “bad” fragments, with the goodness defined according to compressibility. The symbol probabilities change little within fragments and sometimes very little between “close” fragments with different true contexts. While the statistics of individual fragments vary widely, the number of fragments shows little variation. While each fragment corresponds to a context (in some approximate sense) the number of fragments grows quite slowly as the file size increases. While a large file has a few more fragments, these fragments are themselves much larger. They conclude that it is sensible to base contexts on the fragment structure using the current “noisiness” to determine the position with respect to fragments.

**FIGURE 7.10**

Balkenhol and Schtarkov's models.

Symbol	Code
0_a	0 0
0_b	0 1
1	1 0
2–7	1 1 0 b_2 b_1 b_0
8–15	1 1 1 0 b_2 b_1 b_0
16–31	1 1 1 1 0 b_3 b_2 b_1 b_0
32–63	1 1 1 1 1 0 b_4 b_3 b_2 b_1 b_0
64–127	1 1 1 1 1 1 0 b_5 b_4 b_3 b_2 b_1 b_0
127–255	1 1 1 1 1 1 1 b_6 b_5 b_4 b_3 b_2 b_1 b_0

FIGURE 7.11

Deorowicz' coding model.

They first used a cascaded model as shown in Fig. 7.10a. It is truncated to the actual maximum symbol so that the coder wastes no code space in allowing for symbols which will never occur. They then use an order-3 context, based on the rank r , according to $\{r = 0, r = 1, r > 1\}$, to determine whether the coding is within a good fragment, within a bad fragment, or between fragments. Their compressor achieves 2.30 bpc, at the time the best result from a Burrows-Wheeler compressor. In later work, Balkenhol and Schtarkov [27] use the hierarchical model shown in Fig. 7.10b. This has a small model with fast adaptation, two progressively larger ones with slower adaptation and a final constant or order(-1) model in the background. This model has been adopted by other workers as noted later.

Deorowicz [17] also examines the statistical coder, producing the coding model shown in Fig. 7.11, which combines aspects of both the cascaded and the structured coding models. The coding is done with binary encoders (each of the b_i represents a single bit), with the code having some resemblance to an Elias γ code [34].

Note that he includes run-length coding and therefore needs the two symbols 0_a and 0_b to encode the lengths.

Table 7.7 Recent Results on Calgary Corpus Files

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progI	progP	trans
Fenwick (unpublished); average = 2.298 bpc													
1.926	2.356	2.012	4.268	2.464	3.765	2.433	2.439	2.387	0.753	2.476	1.697	1.702	1.488
Balkenhol <i>et al.</i> (1998); average = 2.30 bpc													
1.93	2.33	2.00	4.27	2.47	3.79	2.47	2.44	2.39	0.75	2.47	1.70	1.69	1.47
Deorowicz 2000; average = 2.271 bpc													
1.904	2.317	1.983	4.221	2.450	3.737	2.427	2.411	2.369	0.741	2.446	1.678	1.665	1.448

Code Bit	Number of Possible Contexts	Possible Contexts
first bit	6	last char = 0; length of 0-run ≤ 2 last char = 0; length of 0-run > 2 last char = 1; preceding char = 0 last char = 1; preceding char $\neq 0$ last char > 1 ; preceding char = 0 last char > 1 ; preceding char $\neq 0$
second bit (first = 0)	$\log_2(n)$	length of 0-run is context
second bit (first = 1)	2	last char = 0 or last char = 1
successive code bits	254	all previous code bits form a context

FIGURE 7.12

Deorowicz' context generation.

But Deorowicz then treats the sequence of bits in a context c as a Markov chain of order d and uses a statistical estimator; the details of his context generator are shown in Fig. 7.12. The estimator is complex and the reader is referred to his paper for details.

These methods are among the mathematically most complex used in a Burrows–Wheeler compressor; the results are the best known for a Burrows–Wheeler compressor. Table 7.7 shows some of these results, together with those for Balkenhol *et al.* [18] and some unpublished results of the author (shown in the first row), which retained the structured coder but included a reduced alphabet and sticky MTF. It may be concluded that although improved encoders can give certainly better compression, the gain is difficult and expensive to achieve.

7.11 ELIMINATING MOVE-TO-FRONT

Arnavut and Magliveras [20] introduce *inversion frequency vectors* as an alternative to Move-To-Front recoding. We illustrate with the earlier “mississippi” example, which permutes to “iiiiimppssss”. First transmit the alphabet {imps} and then the symbol frequencies {4, 1, 2, 4}. Then send the positions of each symbol by encoding the gaps of available spaces between symbol occurrences. The symbol ordering is unstated, but Wirth [36] finds that symbols should be sent in decreasing frequency. Here we encode in the order {ispmp}, breaking the tie on s and p by lexicographical ordering. The operations are then as shown in Fig. 7.13.

Action	Sym	Vector
Prepare 11 symbols		X X X X X X X X X X X
Send vector for i	i 0 0 0 0	i i i i X X X X X X
Send vector for s	s 3 0 0 0	i i i i X X X s s s s
Send vector for p	p 1 0 0	i i i i X p p s s s s
Fill remaining space with m	m	i i i i m p p s s s s

FIGURE 7.13
Inversion frequency vector coding.

1	Symbol Index	0	1	2	3	4	5	6	7	8	9	10
2	array	p	s	s	m	i	p	i	s	s	i	i
3	FC	i	i	i	i	m	p	p	s	s	s	s
4	CSPV	4	6	9	10	3	0	5	1	2	7	8
5	LBIV	0	0	0	0	4	5	3	6	6	2	2
6	decorrelated vector	0	0	0	0	4	5	2	6	0	4	0

FIGURE 7.14
Arnavut's inversion coding.

The successive numbers of the vectors indicate the number of *available* spaces to skip over before the current symbol is entered; consecutive symbols have 0's from the zero space. Note that the very last symbol (least frequent) can be inferred as it simply fills the still available spaces.

Arnavut later improved and modified his technique of inversion frequencies, with his system of “inversion coding” [30], illustrated in Fig. 7.14 for the string “iiiiimppssss” (the Burrows–Wheeler permutation of “mississippi” used earlier). Despite the similarity in names, these are really two quite different approaches. His inversion frequencies were really a substitute for Move-To-Front coding, whereas inversion coding applies at a much later stage of the process sequence.

Figure 7.14 shows the following:

- Row 1 is an index of the symbols in the second row, placed here only for reference.
- Row 2 contains the permuted symbols from the Burrows–Wheeler transform.
- Row 3 “FC” (first symbol) has the symbols in order as prepared during the first stage of the decoding step. These symbols form the last symbol of the preceding context for the permuted symbols in row 2.
- Row 4 is Arnavut’s *canonical sorting permutation vector*. Inspection shows that this is neither more nor less than the “link” column of Fig. 7.2, but now 0-origin rather than 1-origin. His coder goes partway into what is traditionally thought of as the *decoding* process.
- Row 5 he calls the *left bigger inversion vector*. The value at LBIV[i] is the number of values in CSPV[0...i-1] which exceed CSPV[i].
- Row 6 is the “decorrelated” values of Row 5. The LBIV entries occur in chunks, one chunk for each symbol. Within each chunk, the values are non-increasing, which allows us to *decorrelate* the values by taking the difference between adjacent values of the same chunk. It is this decorrelation vector which is actually transmitted. The vertical lines here divide the sequence into blocks, one block for each symbol. The initial values in each block are

Table 7.8 Wirth’s and Other Recent Results on Calgary Corpus Files

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progL	progP	trans
Balkenhol <i>et al.</i> (1998); average = 2.30 bpc													
1.93	2.33	2.00	4.27	2.47	3.79	2.47	2.44	2.39	0.75	2.47	1.70	1.69	1.47
Deorowicz (2000); average = 2.271 bpc													
1.904	2.317	1.983	4.221	2.450	3.737	2.427	2.411	2.369	0.741	2.446	1.678	1.665	1.448
Wirth (2001); average = 2.349 bpc													
1.990	2.330	2.012	4.390	2.487	3.811	2.514	2.492	2.424	0.743	2.518	1.763	1.792	1.622

rare (only one for each symbol) and can be transmitted by almost any convenient manner without affecting the final compression.

Arnavut’s technique therefore breaks the sequence

```
forward_permutation — encode — compress — decode
— inverse_permutation
```

at a different point from what is usual, placing the statistical encode–decode step in the middle of the `inverse_permutation` rather than between `encode` and `decode`. Using Balkenhol’s hierarchy as described in the previous section [18] Arnavut achieves a compression of 2.30 bpc, but with outstanding results on the large text files of the Canterbury Corpus.

In a quite different approach, Wirth (see Refs. [36, 37]) abandons the Move-To-Front operation completely and attacks the permuted data directly using techniques taken from PPM compressors. His compressor divides the permuted output into “segments” of like symbols and processes the segments with a compressor based on that of Balkenhol and Schtarkov, employing conditioning and exclusion as from PPM compressors. Run-lengths (the length of a segment) are encoded separately.

Wirth’s results may be compared with those of other recent workers, copied from Fig. 7.8 earlier and combined in Table 7.8. These results demonstrate that Burrows–Wheeler compression without the Move-To-Front stage can certainly achieve good performance.

7.12 USING THE BURROWS–WHEELER TRANSFORM IN FILE SYNCHRONIZATION

Tridgell [35] designed his program `rsync` to synchronize widely separated files by transmitting only changes and differences. Such differences can be detected by checksumming blocks of the files, comparing checksums, and converging on areas of difference. An immediate problem arises with line breaks, which may be sometimes CR, sometimes LF, and sometimes a CR–LF pair, depending on the computer. Thus text blocks which are visually and functionally identical may indicate frequent differences and require a great deal of unnecessary synchronization. Tridgell solves this problem by applying a Burrows–Wheeler transform to both files and then comparing and synchronizing the permuted data. The permutation has the effect of sweeping the CR, LF, or other separators into a few isolated areas of the files, leaving the text areas largely unchanged. Differences due to line breaks are concentrated in a few blocks which are relatively inexpensive to reconcile. He quotes an example where applying the Burrows–Wheeler transform gives a speed-up (effectively a reduction of synchronization traffic) of 27 times (see p. 78, Table 4.5, of [35]).

7.13 FINAL COMMENTS

An enormous amount of work has been done on Burrows–Wheeler compression in the short time since it was introduced, perhaps more than for any other compression technique in a comparable interval. It is new, it is different, it is conceptually simple, and with relatively little effort it gives excellent results. The initial observation by Burrows and Wheeler that “it gives PPM compression with Ziv–Lempel speed” is certainly justified. (Compare this with Effros’ claim, just 6 years later, to combine “PPM compression with BWT complexity”; Burrows–Wheeler compression has certainly arrived!)

Despite all of this work, however, the author feels that Burrows–Wheeler compression is still not *really* understood. The suspicion remains that the Move-To-Front stage is an embarrassing and unnecessary complication, as supported by the work by Arnavut and Wirth. There is undoubtedly some sort of context information in the BWT output, which has been used with some difficulty by Balkenhol *et al.* [18], Deorowicz [17], and Wirth [36]; but is there more, and do we have to find it in some other way?

The improvements so far seem to be largely those of chipping away at a monolithic structure in the hope of getting occasional improvement. While some of the work is solid research with a good theoretical grounding, much other work (and the author includes some of his own in this category!) is little better than optimistic “hacking,” some of which works. The difficulty of improvement is seen in that Fenwick obtained a Calgary Corpus result of 2.52 bpc with a trivial single-model coder and improved this with little trouble to 2.34 bpc (a 7% improvement). Later work has improved that to about 2.27 bpc, a further improvement of 3%. In practical terms this change is negligible.

But this pessimistic tone is hardly justified as an end to the chapter. Burrows–Wheeler compression *does* work. It is competitive in compression with all but the very best statistical compressors. And it achieves this compression with relatively few memory and processor resources, far less than most top-of-the-line PPM methods. The pessimism of the previous paragraph should perhaps be reinterpreted as a call for yet more research on a challenging topic. In summary the algorithm works and it works well, with prospects for future improvement.

7.14 RECENT DEVELOPMENTS

This section presents some Burrows–Wheeler results which became available between the preparation of the chapter and its going to press around July 2002. Interestingly, the four papers represent different approaches, two examining the recoding stage and two the final statistical coder. Quite clearly, the topic is far from worked out. Results for the Calgary Corpus are shown in Table 7.9.

Arnavut [38] continues his development of inversion ranks as an alternative to MTF recoding. With this paper he achieves compression comparable with the best MTF coders, showing that inversion ranks are a viable option.

Balkenhol and Shtarkov [27] concentrate on the final coder, improving their previous context models.

Deorowicz [39] makes further modifications to the MTF recoder, using a “weighted frequency count” to control movement in the ordered list of symbols. This paper reviews other recent alternatives to MTF coding.

Fenwick [40] simplifies the final coder by using “variable-length integers” or “universal codes” instead of the conventional arithmetic or dynamic Huffman coders. Even with this extreme simplification he still obtains compression within about 15% of the best results.

Table 7.9 Recent Burrows–Wheeler Compression Results

File	Arnavut BSIC [38]	Balkenhol BS99 [27]	Deorowicz WFC [39]	Fenwick [40]
bib	1.96	1.91	1.90	2.15
book1	2.22	2.27	2.27	2.81
book2	1.95	1.96	1.96	2.31
geo	4.18	4.16	4.15	5.27
news	2.45	2.42	2.41	2.71
obj1	3.88	3.73	3.70	4.05
obj2	2.54	2.45	2.41	2.57
paper1	2.45	2.41	2.40	2.68
paper2	2.36	2.36	2.35	2.70
pic	0.70	0.72	0.72	0.83
progc	2.50	2.45	2.43	2.68
progl	1.74	1.68	1.67	1.84
propg	1.74	1.68	1.67	1.83
trans	1.55	1.46	1.45	1.58
Average	2.30	2.26	2.25	2.57

7.15 REFERENCES

1. Burrows, M., and D. J. Wheeler, 1994. A Block-Sorting Lossless Data Compression Algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA. Available at gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.
2. Fenwick, P. M., 1995. Experiments with a Block Sorting Text Compression Algorithm, The University of Auckland, Department of Computer Science Report No. 111, May 1995.
3. Fenwick, P. M., 1995. Improvements to the Block Sorting Text Compression Algorithm, The University of Auckland, Department of Computer Science Report No. 120, August 1995.
4. Fenwick, P. M., 1996. Block Sorting Text Compression—Final Report, The University of Auckland, Department of Computer Science Report No. 130, April 1996.
5. Fenwick, P. M., 1996. Block Sorting Text Compression. In *ACSC-96 Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 1996*. pp. 193–202. Available at <ftp.cs.auckland.ac.nz>.
6. Fenwick, P. M., 1996. The Burrows–Wheeler transform for block sorting text compression—Principles and improvements. *The Computer Journal*, Vol. 39, No. 9, pp. 731–740.
7. Seward, J., 1996. Private communication. For notes on the released BZIP see <http://hpux.cae.wisc.edu/man/Sysadmin/bzip-0.21>.
8. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*, Prentice Hall, Englewood Cliffs, NJ. The Calgary Corpus can be found at <ftp://ftp.cs.ucalgary.ca/pub/projects/text.compression.corpus>.
9. Arnold, R., and T. Bell, 1999. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the IEEE Data Compression Conference*, March 1999, pp. 201–210. IEEE Comput. Soc., Los Alamitos, CA. The Canterbury corpus can be found at <http://corpus.canterbury.ac.nz>.
10. Wheeler, D. J., 1995. Private communication. October 1995. [This result was also posted to the *comp.compression.research* newsgroup. Available by anonymous FTP from <ftp.cl.cam.ac.uk/users/djw3>].
11. Bentley, J. L., D. D. Sleator, R. E. Tarjan, and V. K. Wei, 1986. A locally adaptive data compression algorithm. *Communications of the ACM*, Vol. 29, No. 4, pp. 320–330, April 1986.

12. Witten, I., R. Neal, and J. Cleary, 1987. Arithmetic coding for data compression. *Communications of the ACM*, Vol. 30, pp. 520–540.
13. Fenwick, P. M., 1996. A new data structure for cumulative frequency tables: An improved frequency-to-symbol algorithm. *Software—Practice and Experience*, Vol. 26, No. 4, pp. 399–400, April 1996.
14. Moffat, A., R. M. Neal, and I. H. Witten, 1998. Arithmetic coding revisited. *ACM Transactions on Information Systems*, Vol. 16, No. 3, pp. 256–294, July 1998. Source software available from http://www.cs.mu.OZ.AU/~alistair/arith/arith_coder/.
15. Cleary, J. G., and W. J. Teahan, 1997. Unbounded length contexts for PPM. *The Computer Journal*, Vol. 40, No. 2/3, pp. 67–75.
16. Effros, M., 2000. PPM performance with BWT complexity: A fast and effective data compression algorithm. *Proceedings of the IEEE*, Vol. 88, No. 11, pp. 1703–1712, November 2000.
17. Deorowicz, S., 2000. Improvements to Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, Vol. 30, No. 13, pp. 1465–1483, November 2000.
18. Balkenhol, B., S. Kurtz, and Y. M. Shtarkov, 1999. Modifications of the Burrows and Wheeler data compression algorithm. *Proceedings of the IEEE Data Compression Conference*, March 1999, pp. 188–197. IEEE Comput. Soc., Los Alamitos, CA.
19. Chapin, B., and S. R. Tate, 1998. Higher compression from the Burrows–Wheeler transform by modified sorting. *Proceedings of the IEEE Data Compression Conference*, March 1998, p. 532 .IEEE Comput. Soc., Los Alamitos, CA. See also <http://www/cs.unt.edu/home/srt/papers>.
20. Arnavut, Z., and S. S. Magliveras, 1997. Lexical permutation sorting algorithm. *The Computer Journal*, Vol. 40, No. 5, pp. 292–295.
21. Schindler, M., 1997. A fast block sorting algorithm for lossless data compression. *IEEE Data Compression Conference*, March 1997, p. 469. IEEE Comput. Soc., Los Alamitos, CA.
22. Sadakane, K., 1998. A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation. In *IEEE Data Compression Conference*, March 1998, pp. 129–138. IEEE Comput. Soc., Los Alamitos, CA.
23. Seward, J., 2000. On the performance of BWT sorting algorithms. In *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 173–182. IEEE Comput. Soc., Los Alamitos, CA.
24. Shannon, C. E., 1951. Prediction and entropy of printed English. *Bell Systems Technical Journal*, Vol. 30, pp. 50–64.
25. McCreight, E. M., 1976. A space economical suffix-tree construction algorithm. *Journal of the ACM*, Vol. 23, pp. 262–272, April 1976.
26. Larsson, N. J., and K. Sadakane, 1999. Faster suffix sorting. Technical Report, LU-CS-TR:99-214, *Department of Computer Science, Lund University, Sweden*.
27. Balkenhol, B., and Y. M. Schtarkov, 1999. One attempt of a compression algorithm using the BWT. *Technical Report 99-133, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, Universität Bielefeld, Germany*.
28. Manber, U., and E. W. Myers, 1993. Suffix arrays: A new method for online string searches. *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948.
29. Bentley, J. L., and R. Sedgwick, 1997. Fast algorithms for sorting and searching strings. *Proceedings of the 8th Annual ACM-SIAM Symposium of Discrete Algorithms*, pp. 360–369.
30. Arnavut, Z., 2000. Move-to-front and inversion coding. In *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 193–202. IEEE Comput. Soc., Los Alamitos, CA.
31. Larsson, N. J., 1996. Extended application of suffix trees to data compression. In *Proceedings of the IEEE Data Compression Conference*, March 1996, pp. 129–138. IEEE Comput. Soc., Los Alamitos, CA.
32. Fenwick, P. M., 1991. A new technique for self-organizing linear searches. *The Computer Journal*, Vol. 34, No. 5, pp. 450–454.
33. Chapin, B., 2000. Switching between two list update algorithms for higher compression of Burrows–Wheeler transformed data. *Proceedings of the IEEE Data Compression Conference*, March 2000, pp. 183–192. IEEE Comput. Soc., Los Alamitos, CA.
34. Elias, P., 1975. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, Vol. IT-21, No. 2, pp. 194–203, March 1975.

35. Tridgell, A., 1999. *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Department of Computer Science, Australian National University, Canberra, Australia.
36. Wirth, I. W., 2000. *Symbol Driven Compression of Burrows Wheeler Transformed Text*, M.Sc. thesis, Department of Computer Science, Melbourne University, Melbourne, Australia.
37. Wirth, I. W., and A. Moffat, Can we do without ranks in Burrows–Wheeler transform compression?" In *Proceedings of the IEEE Data Compression Conference*, March 2001, IEEE Comput. Soc., Los Alamitos, CA.
38. Arnavut, A., 2002. Generalization of the BWT transformation and inversion ranks. *Proceedings of the IEEE Data Compression Conference*, March 2002, pp. 477–486. IEEE Comput. Soc., Los Alamitos, CA.
39. Deorowicz, S., 2002. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, Vol. 32, No. 2, pp. 99–111.
40. Fenwick, P., 2002. Burrows Wheeler compression with variable-length integer codes. *Software—Practice and Experience*, Vol. 32, No. 13, November 2002.

This Page Intentionally Left Blank

Symbol-Ranking and ACB Compression

PETER FENWICK

8.1 INTRODUCTION

This chapter brings together two compression techniques which are outside the mainstream of accepted methods. Much of this material derives from the author's own work and draws heavily on previously published material.

1. Symbol-ranking compression is a method loosely allied to statistical techniques such as PPM, but is of much more ancient lineage, dating back to Shannon's original work on information.
2. Buynovski's ACB compressor is a relatively new compressor of extremely good performance, but one which does not fit easily into the more conventional taxonomy of compressors.

Both symbol-ranking and ACB compression are described in Salomon's recent book on data compression [1], the only reference known to the author apart from the original papers and reports.

8.2 SYMBOL-RANKING COMPRESSION

Symbol-ranking compression is directly descended from work by Shannon in 1951 in which he first established the information content of English text at 0.6–1.3 bits per letter [2]. He describes two methods. In both a person (the *encoder*) is asked to predict letters of a passage of English text. He postulates an "identical twin" *decoder*, who, being told of the responses, mirrors the actions of the encoder and recovers the original text. In retrospect the application to text compression is obvious, although probably limited by the processing power of the then-available computers, but if this work had been followed up the history of text compression might have been quite different.

Table 8.1 Shannon's Original Prediction Statistics

Guesses, or Symbol Ranking	Probability
1	79%
2	8%
3	3%
4	2%
5	2%

1. In Shannon's first method, the encoder predicts the letter and is then told "correct" or is told the correct answer.
2. In the second method, the encoder must continue predicting until the correct answer is obtained. The output is effectively the position of the symbol in the list, with the sequence of "NO" and the final "YES" responses a unary-coded representation of that rank or position.
3. A third method is a hybrid of the two given by Shannon. After some small number of failures (typically four to six) the response is the correct answer, rather than "NO." This is a useful compromise as predictions, both human and machine, become unreliable and difficult after more than half a dozen attempts.

This algorithm produces a transformation or recoding of the original text, with an output code for every input symbol and a preponderance of small codes. For his Method 2, Shannon gives the results reproduced in Table 8.1.

The distribution is very highly skewed, being dominated by only one value. This implies a low symbol entropy, which in turn implies excellent compressibility.

"All that is needed" to implement a compressor is some algorithm which can produce a symbol list ranked according to the expected probability of occurrence and a following statistical compressor.

8.2.1 Shannon Coder

Figure 8.1 outlines a compressor based on Shannon's second technique, in which the predictor is required to continue until the correct symbol is found.

The heart of the coder is a "predictor" which somehow estimates the next symbol and is then told whether to revise its estimate; the revision instructions (a sequence of "NO" and "YES" responses to the offered symbols) constitute the coder output. The decoder contains an identical predictor which, revising according to the transmitted instructions (and the same contextual and other knowledge), is able to track the coder predictor and eventually arrive at the correct symbol. The prediction is an ordered list of symbols, from most probable to least probable, and the emitted code is the rank of the symbol in this list. The coded output here is a sequence of 1's and 0's: a 0 for each unsuccessful prediction and a 1 for a correct prediction. Both encoder and decoder step to the next symbol in response to a "1" prediction.

In comparison with later compressors, the distinguishing mark of a symbol-ranking compressor is that it encodes *ranks* rather than *symbols* per se or symbol probabilities. In the example just given the encoded data is just a stream of undifferentiated bits. An improvement might be to use a statistical compressor which conditions the relative probabilities according to the rank (the number of immediately preceding 0's) or to condition these probabilities on the actual symbol context. But the work is always in terms of the rank of a symbol, rather than the symbol at that rank.

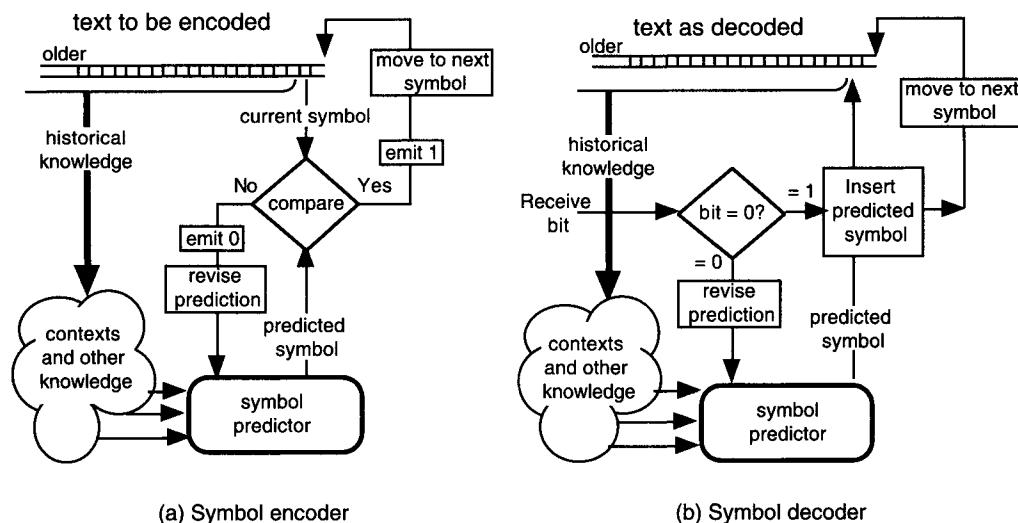


FIGURE 8.1
Shannon's symbol encoder and matching decoder.

8.2.2 History of Symbol-Ranking Compressors

Several compressors can be seen as implementing the methods of symbol-ranking, but with little reference to Shannon's original work.

1. The Move-To-Front compressor of Bentley *et al.* [3] uses words rather than individual characters as its basic compression symbols.
2. Lelewer and Hirschberg [4] use self-organizing lists to store the contexts in a compressor derived from PPM.
3. Howard and Vitter [5] also follow PPM in developing a compressor, but one which explicitly ranks symbols and emits the rank. They show that ranking avoids the need for escape codes to move between orders and also describe an efficient "time-stamp" exclusion mechanism. Much of their paper is devoted to the final encoder, using combinations of quasi-arithmetic coding and Rice codes. Their final compressor has compression approaching that of PPMC, but is considerably faster.
4. The Burrows–Wheeler Transform, described by Burrows and Wheeler [6], uses a context-dependent permutation of the input text to bring together similar contexts and therefore the relatively few symbols which appear in each of those contexts. A Move-To-Front transformation then ranks the symbols according to their recency of occurrence. The Move-To-Front list acts as a good estimate of symbol ranking.

The initial transformations of the Burrows–Wheeler compressor and the symbol recoding of the new method both produce highly skewed symbol distributions. Methods which were developed for the efficient coding of the Burrows–Wheeler compressors are also applicable to symbol-ranking. It will be seen that the frequency distributions of the recoded symbols are very similar for the two cases.

8.2.3 An Example of a Symbol-Ranking Compressor

Bloom [7] described a family of compressors based on the observation that the longest earlier context which matches the current context is an excellent predictor of the next symbol. His compressors follow Shannon's first method in that he flags a prediction as "correct" or "incorrect"

and follows an incorrect flag by the correct symbol. His method does not guarantee to deliver the most probable symbol, but is quite likely to deliver it or, failing that, will deliver one of the other more probable symbols. His various compressors use different methods of signaling success and of specifying the correct symbol, with different speeds and qualities of compression.

Fenwick [8] extended Bloom's method to produce a "proof of concept" symbol-ranking compressor, implementing the scheme of Fig. 8.1. Although it gives reasonable compression, it is far too slow to be regarded as a production compressor. The visible technique is exactly that of Shannon's second method. The symbol predictor offers symbols in the order of their estimated likelihood and the number of unsuccessful offers is encoded. The decompression algorithm is the obvious converse. An initial statistical decoder recovers the sequence of symbol ranks and the symbol prediction mechanism is then called, rejecting as many estimates as indicated by the rank.

The context-discovery and symbol prediction uses techniques derived from sliding-window LZ-77 parsing and with a fast string-matcher similar to that devised by Gutmann (and described by Fenwick [9]).

The algorithm proceeds in several different stages:

1. The preceding data is searched for the longest string matching the current context (the most recently decoded symbols). The symbol immediately following this string is offered as the most probable candidate. So far this is precisely Bloom's algorithm.
2. If the first offer is rejected, the search continues at the original order, looking for more matches which are *not* followed by already offered symbols. As the search proceeds, symbols which have been rejected are added to the *exclusion list* as candidates which are known to be unacceptable.
3. When all available contexts have been searched at an order, the order is reduced by 1 and the search repeated over the whole of the preceding text, from most recent to oldest. Matches followed by an excluded symbol are ignored and any offered symbol is added to the exclusion list.
4. When the order has dropped to zero, the remaining alphabet is searched, again with exclusions. This copy of the alphabet is kept in a Move-To-Front list, rearranged according to all converted symbols to give some preference to the more recent symbols.

Exclusion is handled by the method of Howard and Vitter. A table, indexed by symbol, holds the context or input index at which the symbol was last offered. When a symbol is first considered as a candidate, its current context is compared with that in the table; if the two match the symbol has been already considered within this context and must be excluded. The context of a non-excluded symbol is then saved before the symbol is offered. Advancing to the next position or context automatically invalidates all exclusion entries because none can match the new position.

8.2.3.1 Compression Results

Results for the compressor are shown in Table 8.2, tested on the Calgary compression corpus [10], using a Burrows–Wheeler compressor as a comparison. The symbol-ranking compressor had a "look-back" buffer of 64K symbols and a maximum order of 20 and used a structured coder as developed for the Burrows–Wheeler compressor.

8.2.3.2 The Prediction Process

Some output from an actual encoding of an early version of this chapter is shown in Fig. 8.2. There is a 2-symbol overlap between the two sequences. The actual text is written in boldface type, with the output value just below it (this is the number of wrong estimates for the symbol).

Table 8.2 Results in Compressing the Calgary Corpus, Values in Bits/Character

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progI	progP	trans
Fenwick's Burrows-Wheeler (1996) average = 2.34 bpc													
1.946	2.391	2.039	4.500	2.499	3.865	2.457	2.458	2.413	0.767	2.495	1.715	1.702	1.495
Symbol-ranking, 64K, maxOrder = 20, 2.57 bpc													
2.274	3.028	2.485	5.507	2.841	3.793	2.547	2.591	2.686	0.84	2.551	1.702	1.688	1.503

4	c	i	t										
W	a	s	p									v	
	t	t	s									i	
F	i	l	c									g	
3	r	b c	l									a	
E	s	d l	r									p	
H	p	a r	b									d	
A	⊗	⊗ n	⊗ o									' r	
text	T	h	e	o	u	t	p	u	t	o	f	t	h
output	8	0	0	0	17	0	0	0	0	0	10	4	0
order	2	3	4	5	4	5	6	7	8	9	10	2	3
v			f									k	
i			l									- t s	
g	d	o	"									- l g	
a	s	"	s	d								, c t	
p	.	f	t	c a r								d . v d	i
d		a	d l c	s l s	s x s p n								t
r	i	,	a i	b p v	f e i							n ⊗ y	⊗ s b
text	c	o	d	e	r	h	a	s	a	p	r	e	o
output	7	0	0	1	5	1	5	0	0	0	2	2	7
order	8	9	10	4	3	3	4	5	6	7	5	4	4

FIGURE 8.2

Illustration of symbol-ranking coder symbol prediction.

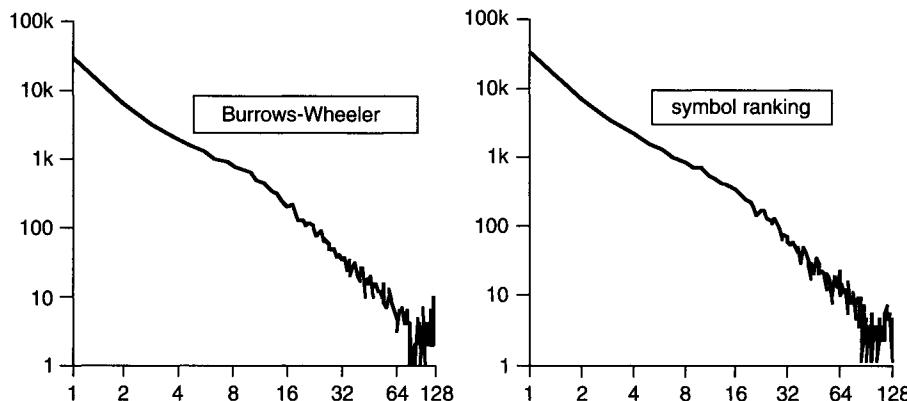
Above each symbol are the predictions for that symbol, with the first always at the top. The eighth and subsequent bad estimates are replaced by a single \otimes . Below the output code is the order at which that code is determined, often with an obvious relation to the preceding text.

What is not so easily conveyed is how the order changes during prediction. For example, in predicting the final "e" of "Shannon code," the unsuccessful "i" is predicted at order 11, but the next prediction (successful) is at order 4, with no external indication of the change in order.

While there may be some difficulty in establishing a symbol, the correct text often proceeds with no trouble for several symbols. A human predictor would get "of" with little difficulty and should also get "Shannon" almost immediately from the overall theme. Again, the latter part of "preponderance" should be predictable (there is no other reasonable word "prepon . . ."). The prediction is often almost eerily like that expected from a person, but one with a limited vocabulary and largely ignorant of idiom.

Table 8.3 Code Rank Frequencies for Block Sorting and Symbol-Ranking Compression

Rank	0	1	2	3	4	5	6	7	8	9
Burrows-Wheeler	58.3%	11.3%	5.5%	3.7%	2.8%	2.3%	1.8%	1.6%	1.4%	1.3%
Symbol-ranking	58.9%	11.6%	5.8%	3.8%	2.7%	2.0%	1.6%	1.4%	1.2%	1.1%

**FIGURE 8.3**

Frequencies of different code ranks for block sorting and symbol-ranking compression.

8.2.3.3 Burrows–Wheeler and Symbol-Ranking

Both the Burrows–Wheeler compressor and the symbol-ranking compressor produce a sequence of symbol ranks as input to the final compressor. Figure 8.3 shows the frequencies of the symbol ranks for the file PAPER1 using the two methods. At this scale the two are essentially identical, except for minor differences for ranks beyond about 16 where the symbol probabilities are quite low. (This figure uses 1-origin ranks to allow a logarithmic display.)

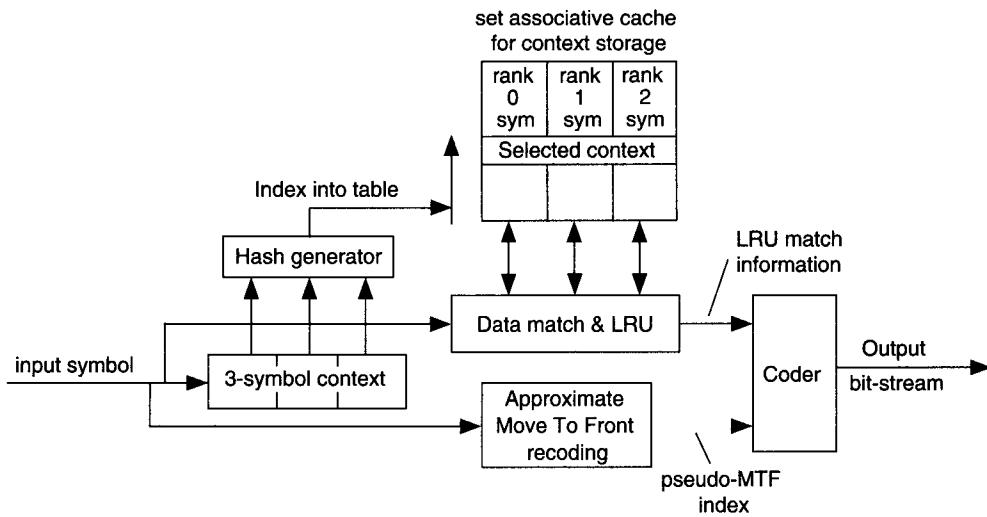
More detail can be seen in Table 8.3, which shows the relative frequencies of symbol ranks for the two compressors, for the relative frequencies greater than about 1%.

The symbol frequencies of Fig. 8.3 approximate a power law ($\text{freq}(n) = n^{-2}$). The exponent is close to 2 for most text files, is higher for more compressible files, and is lower for less compressible files.

8.2.4 A Fast Symbol-Ranking Compressor

Software compressors are often required to give excellent compression, with speed being a lesser consideration, especially if all is in software. An alternative requirement is to provide the best possible speed, even at the cost of good compression. This section describes such a compressor, designed for very fast hardware implementation, which was originally described by Fenwick [11]. It is essentially a cache memory and is probably limited in performance by the speed at which data can be supplied or extracted.

The heart of a symbol-ranking compressor is the management of the lists of ranked symbols for each context. The mechanism here is precisely that of a standard set-associative cache with LRU (Least Recently Used) updating, but used in an unconventional manner. A normal cache is concerned only with hits and misses, with the LRU mechanism controlling symbol replacement but otherwise completely private to the cache and irrelevant to its external operation. Here the LRU functions are crucial, with the position of a hit within the LRU list used to encode a symbol. Figure 8.4 shows the compressor, as it might be implemented in hardware.

**FIGURE 8.4**

A hardware constant-order symbol-ranking compressor.

The three previous symbols are used as an order-3 context, with the 6 low bits of each symbol concatenated as an index to access one line of the cache. (Using a “good” hash function degrades the performance by about 10%, showing that it is best to preserve some of the raw context structure when forming the table index.) The input symbol is matched against the LRU-ordered symbols, the match position is presented to the coder, and the LRU status is updated appropriately, using any standard method such as a permutation vector or physical data shuffling. Earlier versions used 4-way set-association, the 4 bytes fitting conveniently into a 32-bit word, but the 3-way version shown gives very similar performance and is slightly simpler.

The final coding stage accepts the LRU match information and emits it with a defined variable-length coding or with the symbol itself if there is no LRU match. The coding is essentially unary, with a 0 for a correct prediction and a 1 for a failure and step to the next rank. An incoming symbol which matches the most recent member of the LRU set is emitted as a “rank-0” code. A match to the middle member is emitted as “rank-1” and a match to the oldest of the three members of the set as “rank-2.”

Two additional techniques improve compression by about 15%, but make it less relevant for fast hardware implementation. Full details are given in [12] and especially [11], this last including complete code for the compressor.

- A simple and approximate Move-To-Front recoding of the symbol index allows a shorter coding for recent literals.
- Run-length encoding is used for long runs of correctly predicted symbols.

Table 8.4 shows the performance of the compressor, both in the form described and with the two optimizations removed. The compression speed is typically 1 Mbyte per second.

8.3 BUYNOVSKY'S ACB COMPRESSOR

Another recent compressor is the “ACB” algorithm of Buynovsky [13]. It gives excellent compression, the most recent versions of ACB being among the best known text compressors. Its technique is unusual and it does not fit easily into any standard compression taxonomy, although Fenwick [12] did attempt to classify it as a symbol-ranking compressor (or rather a phrase-ranking

Table 8.4 Constant-Order Symbol-Ranking Compressor—64K Contexts, 3 Ranks

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progc	progl	progp	trans
Cache symbol-ranking, 64 K contexts, order 3; average = 3.56 bpc													
3.70	3.94	3.41	6.28	3.86	4.76	3.69	3.63	3.61	1.15	3.60	2.65	2.79	2.76
Cache symbol-ranking, simplified hardware; average = 3.95 bpc													
4.04	4.74	4.00	6.15	4.43	5.02	3.80	4.25	4.32	1.34	4.08	3.03	3.13	3.00

compressor). It may be considered also a development of LZ-77 compressors, with the contents used to limit the coverage of the displacement, much as Bloom does in his LZP compressors [7].

The input text is processed sequentially, with the compressor building a sorted dictionary of all encountered *contexts*. Each symbol processed defines a new context and dictionary entry. Each context is followed in the input text (and in the dictionary) by its corresponding *content* string or phrase. The symbols preceding the current symbol form the *current context*. The current symbol itself and the following symbols form the *current content*. A more conventional compressor might find the best matching context and use that context to predict the next text. ACB recognizes that matching context is probably but one of a group of similar contexts and it is therefore surrounded by a cloud of more or less related contents which may be more appropriate.

When processing a symbol the compressor first finds the longest context matching the current context. This index of the context in the dictionary may be called the *anchor* of the context. (The actual position of the anchor in the input text is irrelevant.) Having found the anchor, the compressor searches neighboring (similar) *contexts* for the phrase from the *contents* with the longest match to the current content. If this best phrase has length λ and occurs a distance δ contexts from the anchor (δ may be negative), the phrase is represented by the couple $\{\delta, \lambda\}$.

The final coding of $\{\delta, \lambda\}$ is rather subtler and largely determines the final quality of the compression. For example, although there is clearly just one *context*, there may be several *contents*, of varying benefit. The benefit of using a longer content phrase must be balanced against the cost of encoding the phrase length and, more importantly, the phrase displacement. Thus sometimes it may be better to specify a short nearby phrase rather than a long remote one while at other times the remote phrase may be preferred.

The operation is illustrated using the dictionary shown in Fig. 8.5 (which is built from an earlier paragraph of this chapter using a case-insensitive sort). The text continues from each “context” to the corresponding “content.” Both context and content are truncated here to about 20 characters. Also note that this example uses only a very short input text and that the selected content may be very poorly related to the current context.

- Text is “is that”; text is processed as far as “... is th” with “at...” to come.
- The best *context* match is “s th”, an order-4 match at index 122 (anchor = 122).
- The best *content* match is “at” of length 2 at index 132 ($\delta = +10, \lambda = 2$).
- Emit the pair $(\delta, \lambda) = (10, 2)$.

And for another example:

- Text is “another thing”; processed to “...her th”, with “ing ...” to come.
- The best context match is “r th”, at 120 or 121, choose smaller index (anchor = 120).
- The best content match is “ing” of length 4 at index 116 ($\delta = -4, \lambda = 4$).
- Emit the pair $(\delta, \lambda) = (-4, 4)$.

index	contexts	contents
115	eding data is search	ed for the longest s
116	longest string match	ing the current cont
117	decoded symbols). Th	e symbol immediately
118	t string matching th	e current context (t
119	deathly following th	is string is offered
120	candidate. So far th	is is precisely Bloo
121	a is searched for th	e longest string mat
122	ing is offered as th	e most probable cand
123	current context (th	e most-recently deco
124	ely Bloom's algorithm	mThe preceding data
125	Bloom's algorithmTh	e preceding data is
126	The preceding data i	s searched for the l
127	lowing this string i	s offered as the mos
128	mbols). The symbol i	mmediately following
129	diate. So far this i	s precisely Bloom's
130	So far this is preci	sely Bloom's algorit
131	algorithmThe precedi	ng data is searched
132). The symbol immedi	ately following this
133	most probable candi	date. So far this is
134	ongest string matchi	ng the current conte
135	iately following thi	s string is offered
136	andidate. So far thi	s is precisely Bloom
137	isely Bloom's algori	thmThe preceding dat
138	following this stri	ng is offered as the
139	for the longest stri	ng matching the curr
140	immediately followi	ng this string is of

FIGURE 8.5

Illustration of ACB compression.

Table 8.5 Results from Buynovski's ABC Compressor

bib	book1	book2	geo	news	obj1	obj2	paper1	paper2	pic	progC	progL	progP	trans
Buynovski's ACB compressor (ACB_2.00c) Average = 2.217 bpc													
1.957	2.348	1.955	4.565	2.344	3.505	2.223	2.352	2.354	0.759	2.342	1.516	1.510	1.304

The coding is subject to various conditioning classes and other optimizations whose details are not disclosed. The result is, however, excellent, as shown in Table 8.5; Buynovski's ACB compressor is one of the best developed to date.

As a final comment on the ACB compressor, note that it encodes phrases into *pairs* of numbers. As there is always some inefficiency in encoding a value into a compact representation, the need to encode two values may be a disadvantage of the ACB method.

ACKNOWLEDGMENTS

Much of the material of this chapter was originally published in the following two papers. The author acknowledges and thanks the publishers for permission to include this material.

1. Springer-Verlag and the editorial board of *Journal of Universal Coding*, for “Symbol Ranking Text Compression with Shannon Recodings” by P. M. Fenwick, 1997, *Journal of Universal Coding*, Vol. 3, No. 2, pp. 70–85.
2. John Wiley for “Symbol Ranking Text Compressors: Review and Implementation” by P. M. Fenwick, 1998, *Software—Practice and Experience*, Vol. 28, No. 5, pp. 547–559, April 1998.

8.4 REFERENCES

1. Salomon, D., 2000. *Data Compression: The Complete Reference*, 2nd Ed., Springer-Verlag, New York.
2. Shannon, C. E., 1951. Prediction and entropy of printed English. *Bell Systems Technical Journal*, Vol. 30, pp. 50–64, January 1951.
3. Bentley, J. L., D. D. Sleator, R. E. Tarjan, and V. K. Wei, 1986. A locally adaptive data compression algorithm. *Communications of the ACM*, Vol. 29, No. 4, pp. 320–330, April 1986.
4. Lelewer, D. A., and D. S. Hirschberg, 1991. Streamlining Context Models for Data Compression, *Data Compression Conference*, pp. 313–322. IEEE Computer Society, Los Alamitos, California.
5. Howard, P. G., and J. S. Vitter, 1993. Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding, *Data Compression Conference*, pp. 98–107. IEEE Computer Society, Los Alamitos, California.
6. Burrows, M., and D. J. Wheeler, 1994. A Block-Sorting Lossless Data Compression Algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994. Available at gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.
7. Bloom, C., 1996. LZP: A New Data Compression Algorithm, *Data Compression Conference*. Vol. 3, No. 2, pp. 70–85, IEEE Computer Society, Los Alamitos, California.
8. Fenwick, P. M., 1997. Symbol ranking text compression with Shannon recodings, *Journal of Universal Computer Science*, Vol. 3, No. 2, pp. 70–85, February 1997.
9. Fenwick, P. M., 1995. Differential Ziv–Lempel text compression. *Journal of Universal Computer Science*, Vol. 1, No. 8, pp. 587–598, August 1995. Available at <http://www.iicm.edu/JUCS>.
10. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*, Prentice Hall, Englewood Cliffs, NJ. The Calgary Corpus can be found at <ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus>.
11. Fenwick, P. M., 1996. A Fast, Constant-Order, Symbol Ranking Text Compressor, Technical Report 145, Department of Computer Science, The University of Auckland, Auckland, New Zealand, April 1996.
12. Fenwick, P. M., 1998. Symbol ranking text compressors: Review and implementation. *Software—Practice and Experience*, Vol. 28, No. 5, pp. 547–559, April 1998.
13. Buynovsky, G., 1994. Associativnoe Kodirovanie. [“Associative Coding,” in Russian], *Monitor, Moscow*, No. 8, pp. 10–19.

PART III

Applications

This Page Intentionally Left Blank

Lossless Image Compression

K. P. SUBBALAKSHMI

OVERVIEW

Lossless image compression deals with the problem of representing an image with as few bits as possible in such a way that the original image can be reconstructed from this representation *without* any error or distortion. Many applications such as medical imaging, preservation of artwork, image archiving, remote sensing, and image analysis call for the use of lossless compression, since these applications cannot tolerate any distortion in the reconstructed images. With the growth in the demand for these applications, there has been a lot of interest lately in lossless image compression schemes. In this chapter, we discuss some of the basic techniques used in lossless image compression.

9.1 INTRODUCTION

An image is often viewed as a two-dimensional array of intensity values, digitized to some number of bits. In most applications 8 bits are used to represent these values, hence the intensity values of a gray-scale image can vary from 0 to 255, whereas in radiology, the images are typically quantized to 12 bits.

It is possible to apply some of the entropy coding schemes developed in the earlier chapters for lossless image compression directly to an image; however, this can be expected to result in only moderate compression. This is because sources like images exhibit significant correlation between adjacent pixels, as can be seen from a visual inspection of a natural image like "Lenna" shown in Fig. 9.1. By directly entropy coding the image, we do not use the substantial structure that exists in the source. Using this inherent structure in the image will let us compress it to a larger extent. Indeed most state-of-the-art lossless image compression techniques exploit this correlation.

**FIGURE 9.1**

Grayscale image “Lenna.”

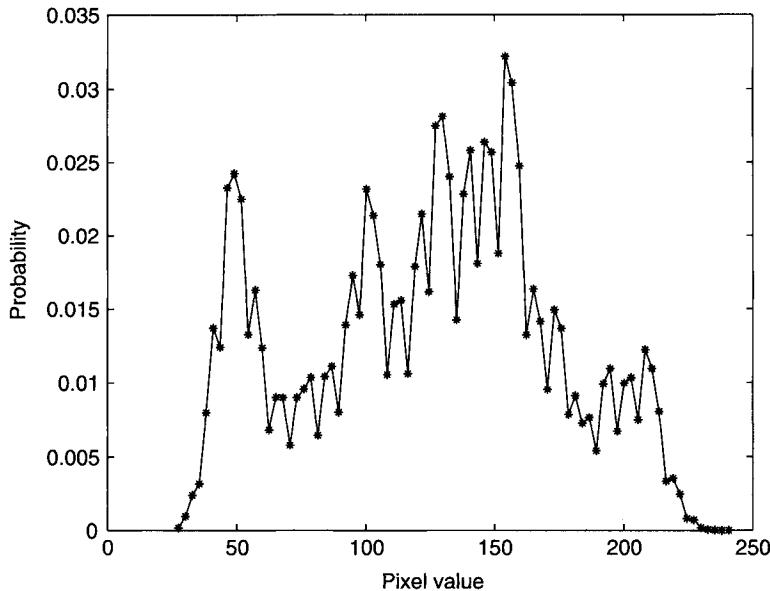
Existing lossless image compression algorithms can be classified broadly into two kinds: those based on spatial prediction and those that are transform based. Both of these approaches can be looked upon as using the correlation between the consecutive pixels in an image to compress it. In prediction-based methods, the current pixel is predicted from a finite context of past pixels and the prediction error is encoded. These techniques are usually applied to the time domain representation of the signal although they could potentially be used in the frequency domain. Transform-based algorithms, on the other hand, are often used to produce a *hierarchical* representation of an image and work in the frequency domain. A hierarchical or *multiresolution* algorithm is based on creating a coarse representation and then adding refinement layers to it. This coarse representation can be used to give a “preview” of the image to the user. Hence such algorithms are very useful in large image database browsing applications, where the coarse resolution version can be used to select the desired image from a huge selection. Hence, transform-based lossless compression has been more popular in applications where a spectrum of compression accuracies (from lossless to lossy) is desired rather than in applications where only lossless compression is preferred. With the success of the latest wavelet-based lossy compression standard, JPEG 2000, transform-based lossless compression methods could become more useful in the future.

This chapter is divided into two parts; the first part introduces some of the basic ideas used in lossless compression and the second delves more deeply into the specific details of algorithms that have been designed for lossless image compression over the years.

9.2 PRELIMINARIES

Entropy coding is more efficient for sources that are distributed less evenly. In other words, when the *probability mass function* (pmf) of a source is skewed or when some of the source outputs occur significantly more often than the others, the source can be more efficiently compressed using entropy codes. Most natural images like Lenna, in Fig. 9.1, do not have such skewed pmf's to begin with (see Fig. 9.2); however, there are ways of extracting sequences with such pmf's from the original images. In sources that are correlated, one of these ways is to predict the current sample value from the past sample values and code the error in prediction. Another way is to predict the pmf of the current pixel based on some of its immediate context. Prediction-based algorithms are discussed in Section 9.2.1.

As mentioned in the Introduction, hierarchical coding of images can be useful in database browsing applications. The basic idea behind this method is to generate several descriptions

**FIGURE 9.2**

Probability mass function for the Lenna image.

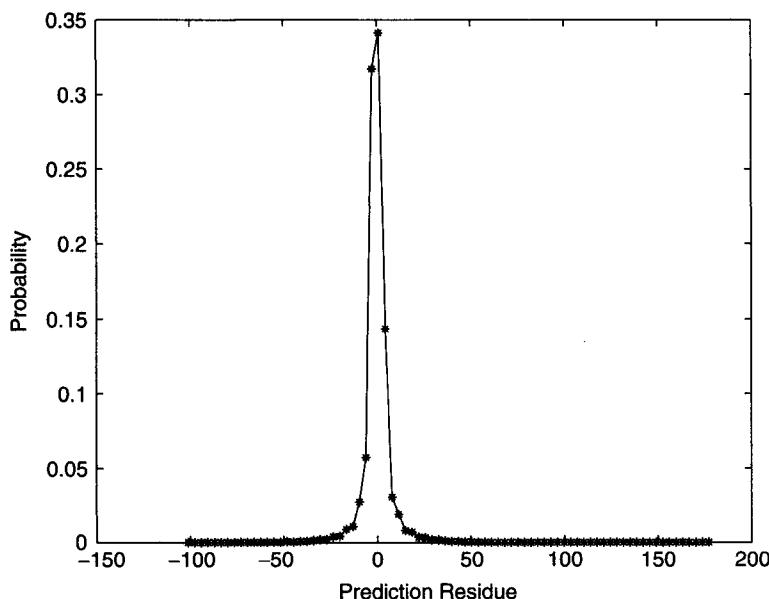
of the image at varying spatial resolutions. The tools used in hierarchical coding methods are discussed in Section 9.2.2.

While predictive and transform-based methods have been used for lossless image compression, some other schemes have also been introduced to enhance the performance of these basic algorithms. These include special scanning techniques that determine the order in which the image is traversed in the coding process. Since scanning orders affect the definition of contexts, they could potentially have a significant effect on the compression performance. Scanning methods used in lossless image coding are discussed in Section 9.2.4.

9.2.1 Spatial Prediction

When there is significant memory between successive samples of a signal, it should be possible to predict the value of any given sample, with a reasonably high degree of accuracy, from some of the preceding samples. The number of samples used in the prediction is largely dependent on the extent of memory in the source and the complexity limits set on the prediction algorithm. Once the first sample (or samples, if more than one preceding sample is used in the prediction) is transmitted, and given that the decoder knows the prediction algorithm, we need transmit only the difference between the prediction and the actual pixel values for the following samples. This is because, given the first sample(s), the decoder will be able to generate its own prediction of the subsequent samples. Adding on the difference values received from the encoder will generate the exact replica of the original image. The difference between the actual signal and the predicted version is often called the *prediction residue* or *prediction error*. In this chapter we will use these terms interchangeably. If the prediction algorithm is reasonably good, then most of the values in the residue will be zero or very close to it. This in turn means that the pmf of the predicted signal will be peaky. The savings in bits comes from the fact that the prediction residues, which make up the bulk of the transmitted signal, can now be compressed well using entropy coding.

A simple predictor for an image is one that uses the previous pixel in the image as the prediction of the current pixel. Formally, if we denote the current pixel by I_n and the previous pixel by I_{n-1} , the prediction \hat{I}_n of I_n is given by $\hat{I}_n = I_{n-1}$. In this case, the prediction error, e_n , is nothing but the difference between the adjacent pixels. Hence, $e_n = I_n - I_{n-1}$. Figure 9.3 shows the pmf of the difference between adjacent pixels for the Lenna image. Comparing this with Fig. 9.2, we can see that even a simple predictor like this can generate a pmf that is significantly skewed in comparison to that of the original image. With more sophisticated predictors, one can expect better gains. Typically, in lossless image compression, the number of samples used in the prediction is more than 1. The pixels used in the prediction are collectively referred to as a *context*. Figure 9.4 shows an example context along with some common notation used to refer to the individual pixels in it.

**FIGURE 9.3**

Probability mass function of the adjacent pixel differences for the Lenna image.

NWNW	NNW	NN	NNE	NENE
WNW	NW	N	NE	ENE
WW	W	In		

FIGURE 9.4

Example context.

Predictors are also classified into *linear* and *non-linear* predictors depending on whether the prediction for the current pixels is calculated as the linear combination of context pixels or not. When the prediction of the current pixel is calculated as the linear combination of the pixels in the context, the resulting predictor is considered to be a linear predictor. Mathematically, the linear prediction \hat{I}_n , of pixel I_n , can be represented by

$$\hat{I}_n = \sum_{i=1}^K \alpha_i I_{n-i}, \quad (9.1)$$

where I_{n-i} , $\forall i \in \{1, \dots, K\}$ refers to pixels in the context and α_i are their corresponding weights in the linear combination. The design of a linear predictor then revolves around the selection of these weights. One approach to finding the weights would be to minimize the mean square prediction error. This, however, does not imply that the entropy of the prediction residue will be minimized.

Non-linear prediction algorithms have been generally less popular for lossless image compression. Some of the non-linear prediction algorithms use classification or learning techniques like neural networks and vector quantization. Rabbani and Dianat [2] used neural networks to predict pixels. Memon *et al.* [3] have looked at the problem of finding optimal non-linear predictors, in which they formulate it as a combinatorial optimization problem and propose some heuristic solutions.

While fixed predictors work well for stationary sources, predictors that *adapt* to the local characteristics of the source are more useful for images since images are essentially non-stationary signals. More details on these types of predictors are discussed in Section 9.3.

9.2.2 Hierarchical Prediction

Spatial predictive coding methods described in the previous section treat the image as a 1-D stream of symbols, whereas in reality, it is not. Considering the image as a 1-D stream imposes a causality constraint in defining a context for any pixel. This is because no pixel that appears after the current pixel in the scan order (see Section 9.2.4 for more on scanning techniques) can be a part of the current pixel's context. This problem can be avoided by using hierarchical or multiresolution prediction techniques that generate more than one representation of the image at different spatial resolutions. These representations can be constructed based on transform, subband, or pyramid coding.

In the basic pyramid coding scheme, the image is divided into small blocks and each block is assigned a representative value. This lower resolution version of the image is transmitted first. Other layers are added later to ultimately construct a lossless version of the image. A simple representation of the block would be its mean; however, more sophisticated schemes using transforms or subband filters are also in vogue. We do not attempt a comprehensive treatment of the theory of transform or wavelet coding here. Excellent sources for more information on this topic include [4–6].

Memon and Sayood [7] classify the hierarchical coding schemes as being either top-down or bottom-up depending on the way they are constructed, while Jung *et al.* [8] classify the existing pyramid-based techniques into either the *mean-based* methods or the *subsample-based* methods. When the average of the block is used as its representation, the pyramid coding scheme is considered to be mean based. In such a scheme a “mean pyramid” is generated where each point is the rounded-off average of non-overlapping image blocks. Once this mean pyramid is constructed, differences between the successive layers of this pyramid are transmitted. If, on the other hand, no

block averaging is done and the differences between successive layers of a pyramid are transmitted as is, the method is referred to as subsample based [8]. Examples of subsample-based pyramid construction can be found in [9–11] and instances of mean-based pyramid construction can be found in [12, 13].

9.2.3 Error Modeling

A key problem in prediction-based algorithms is the efficient representation of the error signal. One of the first prediction-based lossless image coders, the old JPEG standard [14], recommends that the prediction error be encoded using either Huffman codes or arithmetic codes. In the Huffman code option, the error is modeled as a sequence of independent identically distributed (i.i.d.) random variables. In general, however, even sophisticated predictors are unable to produce error signals that can be approximated by i.i.d. random variables. It is common to find some residual structure in the error signals. More sophisticated models are often designed to capture this structure.

The performance of entropy coders depends on the probability model that it uses for the data. If the probability model accurately describes the statistics of the source, then the entropy coder will be more efficient. This makes error or residue modeling [15] very crucial in the performance of the overall algorithm. Most of the error modeling techniques fall under the category of *context modeling* [16] as explained in [17] and applied in [15]. In this approach, the prediction error is encoded with respect to a conditioning state or context, arrived at from the values of previously predicted pixels. Essentially, the role of the error model in this framework is to provide an estimate of the pmf of the error, conditioned on some finite context. This conditional pmf can be estimated by counting the number of occurrences of the symbols within a context [18]. Alternatively, one could assume a model for the pmf of the error and calculate the appropriate parameters. Such an approach was taken in [15] where the pmf of the error was considered to be Laplacian and the variance was calculated. The context used to determine the pmf could be either the same as that used for prediction or different.

The context adaptive lossless image compression algorithm (to be discussed in greater detail in Section 9.3.1), also known as CALIC, chooses a different approach to modeling the prediction error. Here the conditional *expectation* ($E\{e|C\}$) of the error, e , within a context C , is estimated, rather than the conditional *probability*, $\Pr\{e|C\}$, of the error. The rationale for using this approach is based on the fact that the *conditional* mean (as opposed to the *unconditioned* mean) of the error signal is not necessarily zero and that the observed Laplacian distribution without conditioning on the context is a combination of different context-sensitive Laplacian distributions, with different means and variances [16]. Figure 9.5 illustrates this idea with two graphs. The graph labeled (b) represents a more realistic *probability distribution function* (pdf) for the prediction error and is a linear combination of several component Laplacian pdf's shown in the graph labeled (a). $E\{e|C\}$ is used to refine the prediction of the current pixel in CALIC and this process is termed *bias cancellation* by Memon and Wu [16].

9.2.4 Scanning Techniques

In most of the prediction-based lossless image compression techniques, the performance ultimately depends on an estimate of the conditional pmf for a pixel given its past. The past or causal context will depend on the way in which the image is “read” since the causality of the pixels will depend on the scanning method used. Although theoretically, the optimal bit rate at which the image is encoded will not depend on the scan order [19], this is true only if the context is allowed to be infinite. In practice, however, the context is finite and the scan order may be important.

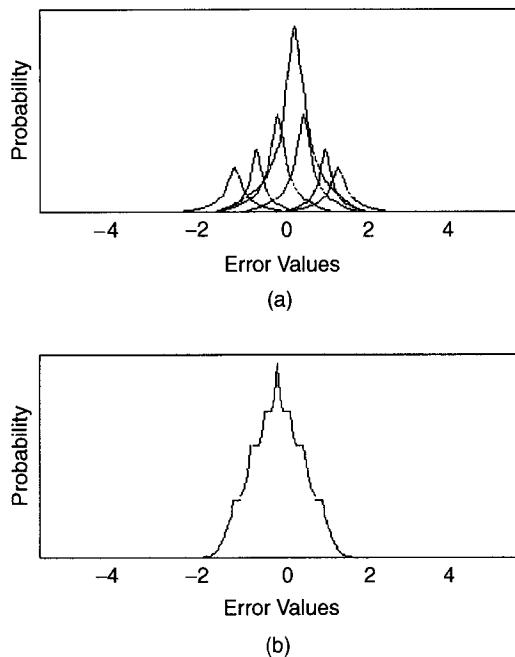


FIGURE 9.5
Distribution of prediction errors.

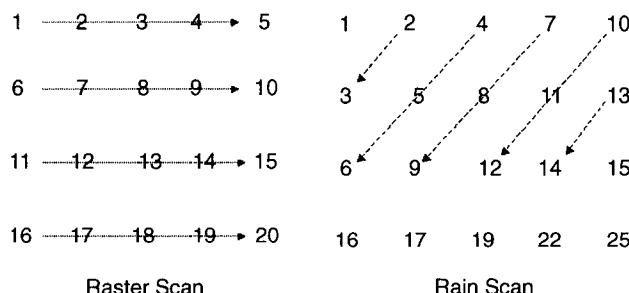


FIGURE 9.6
Scanning patterns.

Although most prediction algorithms use the raster scan order (where the pixels are read from left to right and top to bottom along the direction of the arrows in Fig. 9.6), Lempel and Ziv showed that universal coding of images can be accomplished by using a 1-D encoding method on the image scanned according to the discrete approximation of the Hilbert space-filling curves [20]. The Hilbert scan is shown in Fig. 9.7 for an 8×8 image block. Traversing the block in increasing order of the integer labels gives the Hilbert scan in this figure. Hilbert scanning has been used by many authors for its “clustering” or locality-preserving properties. For example, Perez *et al.* [21] use an adaptive arithmetic code for a one-dimensional stream obtained after Hilbert-scanning multispectral images. Very recently a systematic analysis of scanning patterns was conducted [19] and it was seen that for algorithms that use limited contexts (like most practical algorithms) the Hilbert scan may not lead to significant improvements over the raster scans. Hilbert scans

1	2	15	16	17	20	21	22
4	3	14	13	18	19	24	23
5	8	9	12	31	30	25	26
6	7	10	11	32	29	28	27
59	58	55	54	33	36	37	38
60	57	56	53	34	35	40	39
61	62	51	52	47	46	41	42
64	63	50	49	48	45	44	43

FIGURE 9.7

Hilbert scan.

were also shown to be not very advantageous in increasing run-length, in the work by Quin and Yanagisawa [22]. However, there are other scan orders proposed by certain authors especially as a feature to enhance the performance of their algorithm. One such ordering is the *rain* scan ordering proposed by Seeman *et al.* [23] (also shown in Fig. 9.6).

9.3 PREDICTION FOR LOSSLESS IMAGE COMPRESSION

As mentioned earlier, while static (unchanging) prediction is a good tool for effectively decorrelating a stationary signal, this technique will have to be used with some enhancements when dealing with a non-stationary source. Non-stationary sources like images are characterized by abruptly varying statistics. In order to keep up with this change it would be desirable to make the predictors *adaptive* to local statistics of the image. The adaptive predictors used in lossless image compression can be broadly classified as *switched predictors* or *combined predictors*.

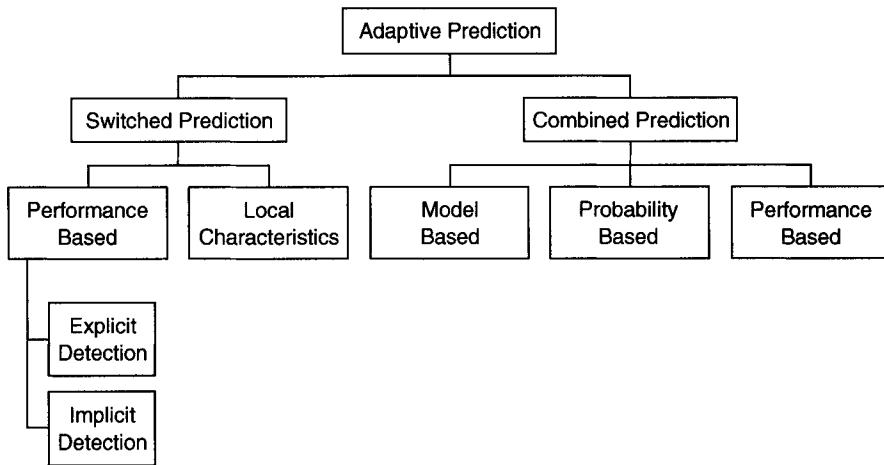
The first class encompasses all techniques where a single subpredictor is chosen from a set. For linear predictors, one might also consider changing the weights of the context pixels in the linear combination according to the local characteristics. This can also be thought of as switched prediction, where the set of subpredictors is calculated somewhat on the fly. In the second category, namely, combined prediction, we do not choose a *single* subpredictor from a set, but rather combine a subset of subpredictors from the entire collection. Figure 9.8 shows a further subclassification of these categories.

9.3.1 Switched Predictors

Linear predictors can be made adaptive by changing the weights in the linear combination of causal neighboring pixels (α_i s in Eq. (9.1)). Alternatively, it is possible to select one predictor from a set of subpredictors that may even be non-linear. Switched prediction encompasses both of the above categories; in fact, the weight-changing strategy may itself be considered a technique of choosing one subpredictor from a repertoire of subpredictors.

The parameters that influence the choice of subpredictor could either be based on the performance of the subpredictor on a set of causal neighbors (performance-based selection) or be based on some local characteristic of the image (local characteristics-based selection).

Sayood and Anderson [24] proposed a switched prediction scheme where the switching is done based on the prediction error performance of the subpredictors. Another simple and effective

**FIGURE 9.8**

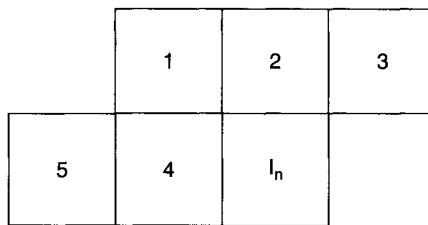
Classification of adaptive prediction schemes.

adaptive predictor was suggested in [25], where the median prediction of a set of subpredictors is calculated and this is used as the actual prediction. A more elaborate prediction scheme that involves the prediction errors for a causal neighborhood of the current pixel was developed in [26]; however, Memon and Sayood [7] contend that for lossless compression this technique does not offer much improvement over the median adaptive predictor.

An image is made up of different regions like edges, smooth surfaces, and textures, all of which have very different characteristics. Predicting pixels that fall in any one of these categories from pixels that belong to other categories can be inefficient. In fact, it may be necessary to change not only the context of the subpredictor, but also the subpredictor's structure according to the local characteristics of the image. Some switched prediction techniques approach this problem by explicitly detecting the presence of edges or gradients and then changing the subpredictor accordingly, while others "detect" these different regions implicitly. Accordingly, the switched predictors based on local characteristics of the image are further classified into those that explicitly detect features and those that implicitly "learn" them (see Fig. 9.8). Some performance-based switched predictors, like the ALCM [27] and JSLUG [28], do not detect edges either explicitly or implicitly. The ALCM and JSLUG include an adaptive predictor that uses a weighted combination of 5 neighborhood pixels. Initially all pixels are assigned equal weights. They are then modified on the fly depending on the prediction error. If the prediction value was higher than the actual pixel value, then the weight of the largest pixel is decremented by $\frac{1}{256}$ and the weight of the smallest pixel in the neighborhood is incremented by the same amount. If the prediction is too low, the exact opposite scheme is used. In the case of ties in the weights of highest (or lowest) pixels, the ties are broken according to the following priority scheme. The pixel labeled 1 in Fig. 9.9 is changed with the highest priority and the pixel labeled 5 with least priority.

9.3.1.1 *Explicit Detection-Based Predictors*

Explicit detection-based predictors detect edges, gradients, or texture regions in images using simple thresholding and some measure of activity in the image. All of the algorithms covered in this section are more recent and were developed in the wake of the call for proposal [29] issued by the International Standards Organization (ISO) in 1994. A comprehensive comparison [16] of all the proposals submitted to the ISO was undertaken by Memon and Wu and some of those algorithms are described in greater detail here.

**FIGURE 9.9**

Priority scheme for resolving ties.

The median edge detection-based (MED) predictor [30] adapts to the local edges which it detects through a simple algorithm. The N , W , and NW pixels (see Fig. 9.4) are used to detect the presence and orientation of a local edge. If a local vertical edge is detected, then the N pixel is used as the prediction of the current pixel. If a horizontal edge is detected, then the W pixel is used for prediction, and if no edge is detected, then the planar predictor is used, where the prediction \hat{I}_n of the pixel I_n is set to $N + W - NW$.

The MED predictor was used in the low-complexity lossless compression for images (LOCO-I) [30, 31], which was Hewlett-Packard's submission to the standardization committee. This algorithm was adopted in the JPEG-LS standard for lossless and near lossless compression.

The CALIC [32] algorithm detects edges using local gradients in the image. CALIC uses the gradient adaptive predictor (GAP), which changes the weights of the context pixels according to the local gradients in the image. The vertical and horizontal gradients are estimated according to the following equations:

$$d_v = |W - NW| + |N - NW| + |N - NE| \quad (9.2)$$

$$d_h = |W - NW| + |N - NN| + |NE - NNE|. \quad (9.3)$$

Once the gradients are estimated, the presence or absence of edges, as well as their orientation, is determined by comparing the difference between the vertical and the horizontal gradients to some experimentally determined thresholds. Then the weights of the pixels used in the predictor are determined. More details can be found in [16]; the CALIC algorithm also uses the bias-cancellation procedure mentioned in Section 9.2.3 as an error feedback mechanism to fine-tune the prediction.

Other prediction schemes in this category include CLARA [33], Mitsubishi's proposal to the standardization effort. CLARA used texture information as well as edge/gradient detection in its prediction.

The study by Memon and Wu [16] found that among the prediction-based algorithms submitted to the ISO, the MED, GAP, and ALCM predictors performed alike in terms of the zeroth order entropy of the prediction error; however, the MED and GAP predictors are computationally less complex than the ALCM predictor.

9.3.1.2 Learning-Based Switched Predictors

Recently, a switched predictor algorithm [34] was designed that uses a clustering-based approach to classify the local characteristics of the image and a gradient descent algorithm to refine an initial set of weights. The algorithm first defines a causal window of pixels for the current pixel and classifies these pixels and their contexts into a given number of clusters. A centroid is then defined for each cluster. Then the current pixel and its context are classified into one of these

clusters (with centroid C_k , say) and then a set of weights ($P_i = \{w_0, w_1, \dots, w_r\}$, where r is the number of pixels in C_k) are picked that minimize the prediction error for the current pixel, using the pixels in C_k . Finally, the gradient descent algorithm is applied to the pixels in C_k so as to refine the set of weights, P_i . The prediction step in this algorithm is rather complex; however, the algorithm performs as well as CALIC for most images and better than CALIC for some [34]. Their experiments suggest that with more sophisticated prediction error encoding, the algorithm can be competitive with CALIC.

The edge-directed prediction algorithm developed by Li and Orchard [35] is another such predictor where the switching is implicit and the local characteristics of the image are learned. This algorithm does not explicitly detect edges, although it uses the fact that the edges are fundamentally different from the non-edge areas. Therefore, the ideal predictor would align the support of the local pmf in the edge area along the edge itself and would thus have a 1-D support. This is intuitively clear if one considers that the best context for any edge pixel would be found ideally along the edge and not in the area away from it. The local pmf of the rest of the image (non-edge parts) will have a 2-D support, since there is no reason to choose context pixels from any one direction over the other.

The best linear predictor for an N th order Markov source $\{x(i)\}$ is given by

$$\hat{x}(i) = E(x(i)|\vec{x}) = \vec{\alpha} \vec{x}^T, \quad \vec{r}_x R_{xx}^{-1}, \quad (9.4)$$

where $\vec{x} = [x(i - n_1) \dots x(i - n_N)]$, $\vec{r}_x = [r_1, r_2, \dots, r_N]$, $r_k = \text{Cov}\{x(i)x(i - n_k)\}$, $\forall k \in \{1, \dots, N\}$ and $R_{xx} = [R_{jk}]$, $R_{jk} = \text{Cov}[x(i - n_j)x(i - n_k)]$, $\forall k, j \in \{1, \dots, N\}$. So, determining the optimal prediction would involve estimating the covariance matrix, R_{xx} , of the signal. When the signal is not stationary, as is the case with images, the above formulation will have to be made adaptive. One way to do this within the above framework is to assume that the image is locally stationary and estimate the covariance matrix of the image based on some causal neighborhood of the current pixel. The estimation of the covariance matrix is done using a least-squares approach in this algorithm by Li and Orchard.

Finally, the predictor coefficients can also be determined using the least-mean-square (LMS) algorithm as in [36] and in one of the subpredictors in the combined predictors algorithm proposed in [37] (see Section 9.3.2).

9.3.2 Combined Predictors

Predictors can also be viewed from a modeling perspective, where each predictor is considered to be a model for the current pixel. A model is usually constructed based on some assumptions about the modeled data. If these assumptions are correct, then the model is said to be “good” for the data. If it is not, then the model will not produce a good prediction of the data. For non-stationary sources, it is difficult to know to what extent the assumptions made by any model are correct. Hence there is an uncertainty about the model to be used. Some authors [37, 38] contend that this uncertainty warrants the use of combined predictors rather than switched predictors. In combined prediction, more than one predictor is used to predict the current pixels. These predictions are then combined to form the final predicted value.

One way of combining subpredictors is to form a linear combination of the prediction values that each of the subpredictors produces. The design issue is then to choose the coefficients of this linear combination. While some authors view the problem as one of modeling (model-based blending), some others prefer a more intuitive approach like penalizing the subpredictors by a term proportional to the prediction error (performance-based blending). Some other authors have also suggested that the *probability distributions* be combined rather than the prediction values (probability blending).

9.3.2.1 Performance-Based Blending of Predictors

Seeman and Tischer proposed a prediction blending algorithm [39] where the coefficients in the linear combination can be used to penalize the subpredictors that result in large prediction errors. Let I_n be the pixel to be predicted. Let $p_j(n - k)$; $j \in \{1, \dots, M\}$ be the j th subpredictor for pixel I_{n-k} . The penalty term for the j th subpredictor is calculated as follows:

$$G_j = \sum_{k=1}^N |I_{n-k} - p_j(n - k)|. \quad (9.5)$$

The prediction for the current pixel is then given by

$$I_n = \frac{1}{D} \sum_{j=1}^M \frac{p_j(n)}{G_j}, \quad (9.6)$$

where D , the normalization factor, is given by $D = \sum_{j=1}^M 1/G_j$. G_j in Eq. (9.5) is computed over a small neighborhood (consisting of M pixels) of the current pixel. This algorithm was tested extensively [39] and it was seen to outperform both MED and GAP in terms of the entropy of the error signal. When coupled with context-based error feedback and entropy coding, it was seen to perform as well as or better than LOCO and almost as well as CALIC for natural images.

Seeman and Tischer then extended their algorithm to another algorithm called the history-based blending of predictors (see Ref. [23]). This subpredictor combination algorithm uses more complex non-linear subpredictors and the mean absolute error (MAE) criterion for blending the predictors rather than the mean squared error (MSE). This is because of the robustness offered by the MAE over the MSE criterion [40]. This algorithm does as well as the CALIC and the LOCO algorithms, albeit at increased complexity [23]. Part of the success of this algorithm has been attributed to the rain order scanning discussed in Section 9.2.4.

In [41, 42], Aiazzi *et al.* introduced a clustering-based combined prediction algorithm. Here, the image is first split into several blocks and the minimum mean square error linear predictor is calculated for each block. These are then classified into M prototype subpredictors. The image is then traversed in raster scan order (Fig. 9.6) and the MSE is calculated for each pixel using each subpredictor. The M subpredictors are then combined using weights proportional to the inverse of their MSEs. More details on how the clustering is done and the actual expression for the weights can be found in [41].

9.3.2.2 Model-Based Blending

As noted earlier, predictors can be viewed as models for a particular pixel. In the Bayesian approach to combining subpredictors, each subpredictor is considered to be a model and a risk is associated with picking each of these models. Bayesian model averaging (BMA) is a coherent mechanism to combine these models. According to BMA, given the data D and a group of models M_k , the Bayesian estimate θ_B of a parameter θ is determined by

$$\theta_B = E[\theta|D] = \sum_{k=1}^K \theta_k \Pr(M_k|D), \quad (9.7)$$

where $\theta_k = E[\theta|M_k]$. It has been demonstrated that BMA provides better predictive ability than choosing any single model [43].

Lee [37] recently proposed a linear combination of subpredictors where the coefficients in the linear combination were determined using a Bayesian approach. In this algorithm five different subpredictors are used, four of which are simple predictors using the previous pixel as the prediction of the current pixel. The fifth subpredictor is adaptive, and the weights of the causal neighbors

in the linear combination are determined using a LMS-based algorithm. The prediction errors of the subpredictors are modeled as Laplacian distributions and the *a posteriori* probability of the prediction error is used as the combination coefficient in the linear blending of predictors. The performance of this algorithm was found to be comparable to that of CALIC [37].

Deng and Ye [44] used a Lagrange multiplier method in the linear combination of subpredictors. If $p_j(n)$ is the prediction of the n th pixel using the j th subpredictor, then $e_j(n)$ is the corresponding prediction error. The linear combination of subpredictors for the n th pixel is given by

$$P(n) = \sum_{j=1}^M \alpha_j(n) p_j(n), \quad (9.8)$$

where $\alpha_j(n)$ are the coefficients in the linear combination of predictors. If we denote the overall prediction error for the combined predictor by $e(i)$, then $e(i) = I(i) - P(i)$, where $I(i)$ is the current pixel and $P(i)$ is its prediction using the combined predictor. The algorithm proposed by Deng and Ye uses the Lagrangian multiplier method

$$F = \sum_i e^2(i) + \beta \left(1 - \sum_{j=1}^M \alpha_j(n) \right), \quad (9.9)$$

to minimize the mean square error of the predictor over a training block of pixels with the condition that the $\alpha_j(n)$ s (Eq. (9.8)) sum to 1. The coefficients of the linear combination are found by solving the following set of equations:

$$\frac{\partial F}{\partial \alpha_j(n)} = 0 \quad j = 1, 2, \dots, M \quad (9.10)$$

$$\frac{\partial F}{\partial \beta} = 0 \quad (9.11)$$

How this algorithm fits in the Bayesian model averaging framework is described in greater detail in [38].

9.3.2.3 Probability Domain Blending

In an entirely different approach to blending subpredictors, Meyer and Tischer proposed an algorithm [45, 46] where the blending took place in the probability domain rather than the spatial domain. In this two-pass algorithm, the first part consists of selecting several parameters (weights of the individual subpredictors) for the encoding in such a way as to produce the shortest encoded stream. The encoding per se takes place in the second pass. Because of the two-pass nature of this algorithm, the resulting compressed image consists of a header followed by the actual encoded image. Apart from the usual pixel predictor, this algorithm uses two other kinds of predictors: one that predicts the magnitude of *prediction error* of the subpredictor based on the prediction errors of a set of causal neighbors and another that predicts how well the pixel predictor will perform on a certain pixel given its performance in a causal neighborhood.

Since the other combined predictors discussed so far produce only a single prediction value, there will be only a single pdf used to ultimately code the pixel. This pdf will be centered around the resulting predicted value. Thus, it will not be possible to model more complex pdf's (like bimodals) if this pdf is itself not complex. So, these methods could potentially suffer along edges. This algorithm gets around this problem by first calculating the pdf of the current pixel using *each* of the subpredictors and then blending the probability density functions rather than the prediction values. More details on the probability model used in this method and the combination of pdf's can be found in [46].

9.4 HIERARCHICAL LOSSLESS IMAGE CODING

As noted in Section 9.2.2, spatial prediction techniques impose a causal structure on the image which is essentially artificial. Hierarchical prediction schemes avoid imposing such a structure on the image by splitting an image into different *layers*, each of which represents the entire image but at different spatial resolutions. One of the first hierarchical coding algorithms to be developed was *hierarchical interpolation* (HINT) [47]. In the HINT algorithm, the pixels marked Δ in Fig. 9.10 are transmitted first using lossless DPCM, then the pixels marked \circ are predicted from the first set and the prediction error is transmitted. Next, the pixels marked X are predicted from the \circ and Δ pixels and the prediction error is transmitted, and finally, the pixels labeled \star and \bullet are predicted from their neighbors (now known) and the error is transmitted. Linney and Gregson [48] proposed a modified HINT scheme where the subsampling rate was varied according to the local frequency content of the image. Other enhancements to the HINT algorithm can be found in [49, 50].

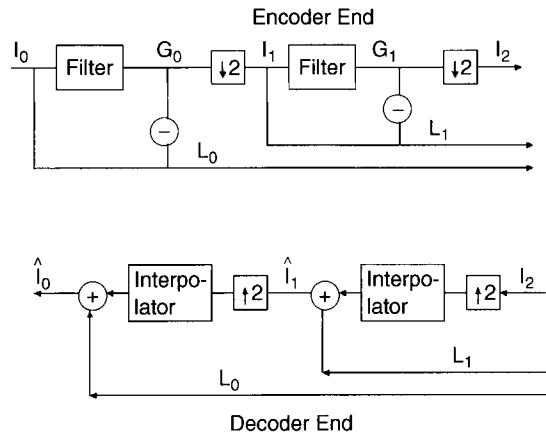
Pyramidal or hierarchical schemes can be created by using both subband and wavelet transforms. In their pioneering work, Burt and Adelson [51] developed the Laplacian pyramidal decomposition of images. In their scheme, the image I_0 is lowpass-filtered to obtain G_0 . The difference between I_0 and G_0 is saved as L_0 , and then G_0 is down-sampled by a factor of 2 along the vertical and horizontal directions to yield I_1 . The process is then applied to the down-sampled version I_1 , to generate G_1 , L_1 , and I_2 . A third level is then generated along the same lines, to yield G_2 , L_2 , and I_3 . The subbands I_3 , L_2 , L_1 , and L_0 are transmitted. The decoder recovers the signal by successive upsampling, interpolation, and addition of difference images. The general pyramid coding scheme for two stages is shown in Fig. 9.11. Although down-sampling in both directions is represented by a *single* block in this figure, it must be remembered that after each such down-sampling, the total number of pixels is reduced to a quarter of the original value.

In the basic pyramid structure, the total number of data values that need to be transmitted for an $N \times N$ image is greater than N^2 . Hence there is an expansion in the number of points that need to be transmitted. Despite this increase, it is possible to encode the mean-residue pyramid using fewer bits than needed to entropy code the image. It is also possible to decrease this expansion in several ways. One of the first schemes to achieve this goal was the *bin-tree* data structure developed to represent an image, along with a way to map pixel pairs to *composite-differentiator* pairs proposed in [52]. Another popular approach that does not cause data expansion is the

Δ	\bullet	x	\bullet	Δ	\bullet	x	\bullet	Δ
\bullet	\star	\bullet	\star	\bullet	\star	\bullet	\star	\bullet
x	\bullet	\circ	\bullet	x	\bullet	\circ	\bullet	x
Δ	\star	x	\star	Δ	\star	x	\star	Δ
\bullet								
x	\star	\circ	\star	x	\star	\circ	\star	x
Δ	\bullet	x	\bullet	Δ	\bullet	x	\bullet	Δ
\bullet	\star	\bullet	\star	\bullet	\star	\bullet	\star	\bullet

FIGURE 9.10

Hierarchical interpolation scheme for decorrelation.

**FIGURE 9.11**

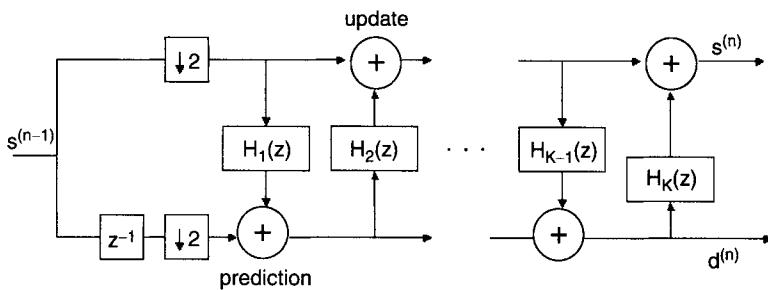
A generic pyramid coding scheme.

reduced-pyramid structure proposed by Wang and Goldberg [53]. A reduced *Laplacian* pyramid structure was proposed in [9]. Other reduced-pyramid structures based on transforms were presented in [13, 54–56].

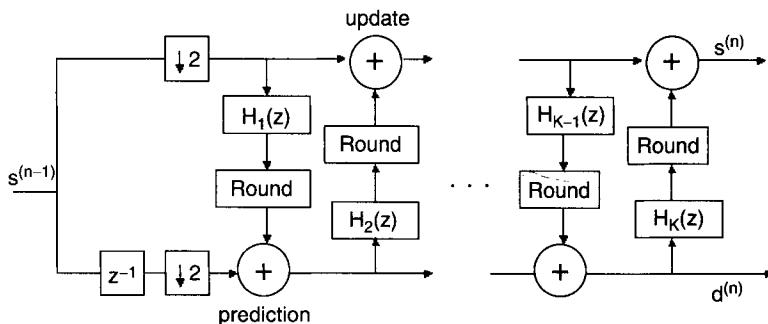
Jung *et al.* proposed a rounding transform [8] that could be used in a lossless pyramid structured coding. This transform maps an integer vector to another by using weighted average and difference filters, followed by a rounding operation, and it generalizes some of the previous pyramid-based schemes like those found in [12, 13, 53]. It was then extended to an overlapping rounding transform [57], which is defined as a two-port FIR filtering system with a pair of rounding operations. Very recently, a method for optimal construction of a class of reduced pyramids was presented in [58], where the interpolation synthesis postfilters are selected to minimize the error variance at each level of the pyramid.

Although transform coding has been very useful in the context of lossy signal compression, there is one significant drawback to using this technique directly for lossless image compression: the transform coefficients are real numbers. Representing real numbers efficiently and without any loss is tricky. Although dyadic representation of the numbers with rescaling can be used, this causes an increase in the dynamic range. In [59–61] a new technique was described, called *lifting*, which allows the transformation of any perfect reconstruction filter to an integer equivalent. Any perfect reconstruction filter bank structure can be transformed to a lifting structure (LS) [59]. For the LS, the signal is split into two streams consisting of the even indexed and odd indexed samples, respectively. One of the streams is passed through a lifting filter and added onto the other stream. The role of the two streams can be reversed and the lifting steps reapplied. Figure 9.12 shows a schematic representation of the lifting steps. After K stages, one of the streams corresponds to the high-pass signal, $d^{(n)}$, and the other corresponds to the low-pass signal, $s^{(n)}$. Lifting steps that use the low-pass signals are called prediction steps and those that use the high-pass signal are called update steps. In Fig. 9.12, the steps that use $H_i(z)$, where i is even, are called the prediction steps. The inverse filter is given by $H'_k(z) = -H_k(z)$. The interesting point about the LS construction is that one could replace the lifting filters with a non-linear step and still have perfect reconstruction. So, introducing a rounding-off operation in the steps, as shown in Fig. 9.13, creates an integer wavelet transform (IWT) where integer inputs produce integer outputs [62]. IWT was applied to lossless image compression in [63].

Several researchers have studied IWT for lossy and lossless image compression [64, 65]. A theoretical framework to quantify the distortion caused by the use of IWT for lossy image

**FIGURE 9.12**

Basic lifting-based wavelet decomposition.

**FIGURE 9.13**

IWT based on the lifting scheme.

compression was presented in [66]. Boulgouris *et al.* [67] constructed optimal filters in the sense of minimizing the variance of the prediction error of a wide sense stationary signal of a lifting in the general n -dimensional case. Cheung *et al.* have designed a filter for use with the lifting scheme based on the box and slope multiscaling system [68]. Other attempts to address the issue of finding efficient filters for use with a lifting scheme include [69], which considers the problem of finding high-performance factorizations of the wavelet filters. A two-layered approach, based on the use of wavelet *packets* in the reversible integer format, was presented in [70], which makes the scheme locally adaptive. Since the codecs based on the discrete wavelet transforms can be used with IWT, there has also been interest in using a similar structure for lossless coding [71–74].

9.5 CONCLUSIONS

An image is characterized by a significant amount of redundancy in the sense that the adjacent pixels in an image are highly correlated. Making use of this correlation forms the crux of the lossless image compression techniques. One of the ways to use this correlation is to predict a pixel from its neighbors. This can be done in the spatial domain using a causal context for the pixel or it can be done hierarchically using pyramid, subband, or transform coding methods. The prediction methods involve transmitting the prediction errors efficiently, usually involving good statistical models for the errors. Lossless compression of images can also be affected by

the scanning techniques that are used in defining the context. This chapter discussed different strategies that exploit the correlation in the image and various error modeling techniques and scanning patterns used in lossless image compression.

9.6 REFERENCES

1. Deleted at proof.
2. Rabbani, M., and S. Dianat, 1993. Differential pulse code modulation compression of images using artificial neural networks. In *SPIE, Image and Video Processing*, pp. 197–203.
3. Memon, N., S. Ray, and K. Sayood, 1994. Differential lossless encoding of images using nonlinear prediction techniques. In *International Conference on Image Processing*, pp. 841–845.
4. Sayood, K., 2000. *Introduction to Data Compression*, 2nd ed. Multimedia Information Systems, San Francisco, CA, and Morgan Kauffman, San Mateo, CA.
5. Akansu, A., and M. Smith (Eds.), 1996. *Subband and Wavelet Transforms Designs and Applications*. Kluwer Academic, Dordrecht/Norwell, MA.
6. Burrus, C., R. Gopinath, and H. Guo, 1998. *Introduction to Wavelets and Wavelet Transforms: A Primer*. Prentice Hall, New York.
7. Memon, N., and K. Sayood, 1995. Lossless image compression—A comparative study. *SPIE Proceedings, Still Image Compression*, Vol. 2418, pp. 8–20.
8. Jung, H.-Y., T.-Y. Choi, and R. Prost, 1998. Rounding transform and its application to lossless pyramid structured coding. *IEEE Transactions on Image Processing*, Vol. 7, pp. 234–237, February 1998.
9. Aiazi, B., L. Alparone, and S. Barontini, 1996. A reduced Laplacian pyramid for lossless and progressive image communication. *IEEE Transactions on Communications*, Vol. 44, pp. 18–23, January 1996.
10. Houlding, D., and J. Vaisey, 1995. Low entropy image pyramids for efficient lossless coding. *IEEE Transactions on Image Processing*, Vol. 4, pp. 1150–1153, August 1995.
11. Roos, P., M. Viergever, M. V. Dijke, and J. Peters, 1998. Reversible intraframe compression of medical images. *IEEE Transactions on Medical Imaging*, Vol. 4, pp. 328–336.
12. Blume, H., and A. Fand, 1989. Reversible and irreversible image data compression using the s —transform and Lempel Ziv coding. *SPIE Medical Imaging III: Image Capture and Display*, Vol. 1091, pp. 2–18.
13. Kim, W., P. Balsara, D. H., III, and J. Park, 1995. Hierarchy embedded differential images for progressive transmission using lossless compression. *IEEE Circuits and Systems for Video Technology*, Vol. 5, pp. 1–13, February 1995.
14. Pennebaker, W., and J. Mitchell, 1993. *JPEG Still Image Compression Standard*. Van Norstrand Reinhold, New York.
15. Howard, P., and J. Vitter, 1992. Error modeling for hierarchical lossless image compression. In *Data Compression Conference*, pp. 269–278.
16. Memon, N., and X. Wu, 1998. Recent developments in lossless image compression. *The Computer Journal*, Vol. 40, pp. 117–126, June 1998.
17. Rissanen, J., and G. Langdon, 1981. Universal modeling and coding. *IEEE Transactions on Information Theory*, Vol. 1, pp. 12–22, January 1981.
18. Todd, S., G. Langdon, and J. Rissanen, 1985. Paramter reduction and context selection for compression of gray scale images. *IBM Journal of Research and Development*, Vol. 29, No. 2, pp. 188–193.
19. Memon, N., D. L. Neuhoff, and S. Shende, 2000. An analysis of some common scanning techniques for lossless image compression. *IEEE Transactions on Image Processing*, Vol. 9, pp. 1837–1848, November 2000.
20. Lempel and Ziv, 1986. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, Vol. 32, No. 1, pp. 1–8.
21. Perez, A., S. Kamata, and E. Kawaguchi, 1991. Arithmetic coding model for compression of landsat images. *Visual Communications and Image Processing*, Vol. SPIE 1605, pp. 879–884.
22. Quin, A., and Y. Yanagisawa, 1989. On data compaction of scanning curves. *Computer Journal*, Vol. 32, No. 6, pp. 563–566.

23. Seeman, T., P. Tischer, and B. Meyer, 1997. History-based blending of image subpredictors. In *Proceedings of the Picture Coding Symposium*, Berlin, Germany, pp. 147–151.
24. Sayood, K., and K. Anderson, 1992. A differential lossless image compression algorithm. *IEEE Transactions on Signal Processing*, Vol. 40, No. 1, pp. 236–241.
25. Martucci, S., 1990. Reversible compression of HDTV images using median adaptive prediction and arithmetic coding. In *IEEE, International Symposium on Circuits and Systems*, pp. 1310–1313.
26. Zschunke, W., 1977. DPCM picture coding with adaptive prediction. *IEEE Transactions on Communications*, Vol. 25, No. 11, pp. 1295–1302.
27. Speck, D., 1996. Fast robust adaptation of predictor weights from min/max neighbouring pixels for minimum conditional entropy. In *Twenty-ninth Asilomar Conference on Signals, Systems and Computers*, 1997, Asilomar, pp. 234–238.
28. Langdon, G., D. Speck, C. Haidinsky, and S. Macy, 1995. Contribution to JTC 1.29.12: JSLUG, ISO Working Document ISO/IEC JTC1/SC29/WG1 N199.
29. ISO/IEC JTC 1/SC 29/WG1, 1994. Call for Contributions—Lossless Compression of Continuous Tone Still Pictures. ISO Working document ISO/IEC JTC1/SC29/WG1/N196.
30. Weinberger, M., G. Seroussi, and G. Sapiro, 1995. LOCO-I: A Low Complexity Lossless Image Compression Algorithm, ISO Working document ISO/IEC JTC1/SC29/WG1 N281, August 1995.
31. Weinberger, M., G. Seroussi, and G. Sapiro, 2000. LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS. *IEEE Transactions on Image Processing*, Vol. 9, pp. 1309–1324, August 2000.
32. Wu, X., N. Memon, and K. Sayood, 1995. A Context Based, Adaptive, Lossless/Near-Lossless Image Coding Scheme for Continuous-Tone Images, ISO Working document ISO/IEC JTC1/SC29/WG1/N256.
33. Ueno, I., and F. Ono, 1995. CLARA: Continuous-Tone Lossless Coding with Edge Analysis and Range Amplitude Detection, ISO Working document ISO/IEC JTC1/SC29/WG1/N197.
34. Motta, G., J. Storer, and B. Carpentieri, 2000. Lossless image coding via adaptive linear prediction and classification. *Proceedings of the IEEE*, Vol. 88, pp. 1790–1796, November 2000.
35. Li, X., and M. T. Orchard, 1998. Edge directed prediction for lossless compression of natural images. In *IEEE International Symposium on Circuits and Systems*, pp. 58–62.
36. Chung, Y., and M. Kanefsky, 1992. On 2-D recursive LMS algorithm using the ARMA prediction for ADPCM encoding of images. *IEEE Transactions on Image Processing*, Vol. 1, pp. 416–442.
37. Lee, W., 1999. Edge adaptive prediction for lossless image coding. In *Data Compression Conference*, pp. 483–490.
38. Deng, G., H. Ye, and L. Cahill, 2000. Adaptive combination of linear predictors for lossless image compression. *IEE Proceedings—Scientific Measurements and Technology*, Vol. 147, pp. 414–419, November 2000.
39. Seemann, T., and P. Tischer, 1997. Generalized Locally Adaptive DPCM, Technical Report 97/301, Monash University, Australia.
40. Tischer, P., 1994. Optimal Predictors for image Compression, Technical Report 94/189, Monash University, Australia.
41. Aiazi, B., S. Baronti, and L. Alparone, 2000. Near lossless compression using relaxation labeled prediction. In *International Conference on Image Processing*, Vancouver, British Columbia, Canada, pp. 148–151.
42. Aiazi, B., S. Baronti, and L. Alparone, 1999. Lossless image compression based on an enhanced fuzzy regression prediction. In *International Conference on Image Processing*, Berlin, Germany, pp. 435–439.
43. Hoeting, J., D. Madigan, A. Raftery, and C. Volinsky, 1998. Bayesian model averaging: A tutorial, Technical Report 9814, Department of Statistics, Colorado State University, Fort Collins.
44. Deng, G., and H. Ye, 1999. Lossless image compression using adaptive predictor combination, symbol mapping and context filtering. In *International Conference on Image Processing*, Kobe, Japan, Vol. 4, pp. 63–67.
45. Meyer, B., and P. E. Tischer, 1997. TMW—A new method for near lossless compression of greyscale images. In *Picture Coding Symposium*, Berlin, Germany, pp. 533–538.
46. Meyer, B., and P. Tischer, 1998. Extending tmw for lossless image compression. In *Data Compression Conference*, pp. 458–470.

47. Endoh, T., and Y. Yamazaki, 1984. Progressive coding scheme for interactive image communications. In *Global Communications Conference*, Vol. 3, pp. 1426–1433.
48. Linney, N., and P. Gregson, 1994. HINT based multiresolution image compression. In *Canadian Conference on Electrical and Computer Engineering*, Vol. 2, pp. 580–583.
49. Ramabadran, T., and K. Chen, 1992. The use of contextual information in the reversible compression of medical images. *IEEE Transactions on Medical Imaging*, Vol. 11, No. 2, pp. 185–195.
50. Howard, P., and J. Vitter, 1991. New methods for lossless image compression using arithmetic coding. In *Data Compression Conference*, pp. 257–266.
51. Burt, P., and E. Adelson, 1983. Laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, Vol. Com-31, pp. 532–540, April 1983.
52. Knowlton, K., 1980. Progressive transmission of grey-scale and binary pictures by simple, efficient and lossless encoding schemes. *Proceedings of the IEEE*, Vol. 68, No. 7, pp. 885–896.
53. Wang, L., and M. Goldberg, 1989. Reduced-difference pyramid: A data structure for progressive image transmission. *Optical Engineering*, Vol. 28, pp. 708–716, July 1989.
54. Rabbani, M., and P. Jones, 1991. *Digital Image Compression Techniques*, Vol. TT7, Tutorial Texts Series. Int. Soc. Opt. Eng., Bellingham, WA.
55. Said, A., and W. Pearlman, 1993. Reversible image compression via multiresolution representation and predictive coding. In *Visual Communications and Signal Processing*, pp. 664–673.
56. Zandi, A., J. Allen, E. Schwartz, and M. Boliek, 1995. CREW: Compression by reversible embeddded wavelets. In *Data Compression Conference*, pp. 664–673.
57. Jung, H., and R. Prost, 1997. Rounding transform based approach for lossless subband coding. In *International Conference on Image Processing*, Santa Barbara, CA, Vol. 2, pp. 271–278.
58. Tzovaras, D., and M. Strintzis, 2000. Optimal construction of reduced pyramids for lossless and progressive image coding. *IEEE Transactions on Circuits and Systems.II. Analog and Digital Signal Processing*, Vol. 47, pp. 332–348, April 2000.
59. Sweldens, W., 1996. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Journal of Applied and Computer Harmonic Analysis*, Vol. 3, pp. 186–200.
60. Daubechies, I., and W. Sweldens, 1996. Factoring Wavelet Transforms into Lifting Steps, Technical Report, Bell Laboratories, Lucent Technologies.
61. Daubechies, I., and W. Sweldens, 1998. Factoring wavelet transforms into lifting steps. *Journal of Fourier Analysis and Applications*, Vol. 4, No. 3, pp. 245–267.
62. Calderbank, R., I. Daubecheis, W. Sweldens, and B-L. Yeo, 1998. Wavelets that map integers to integers. *Applied Computer Harmonic Analysis*, Vol. 5, No. 3, pp. 332–369.
63. Calderbank, R., I. Daubecheis, W. Sweldens, and B-L. Yeo, 1997. Lossless image compression using integer to integer wavelet transform. In *International Conference on Image Processing*, Vol. I, pp. 596–599.
64. Sheng, F., A. Bilgin, P. Sementilli, and M. Marcellin, 1998. Lossy and lossless image compression using reversible integer wavelet transforms. In *International Conference on Image Processing*, Los Alamitos, CA, Vol. 3, pp. 876–880.
65. Adams, M., and Kossentini, 2000. Reversible integer to integer transforms for image compression: Performance evaluation and analysis. *IEEE Transactions on Image Processing*, Vol. 9, pp. 1010–1024, June 2000.
66. Reichel, J., G. Menegaz, M. Nadenau, and M. Kunt, 2001. Integer wavelet transform for embedded lossy to lossless image compression. *IEEE Transactions on Image Processing*, Vol. 10, pp. 383–392, March 2001.
67. Boulgouris, N., D. Tzovaras, and M. Strintzis, 2001. Lossless image compression based on optimal prediction, adaptive lifting and conditional arithmetic coding. *IEEE Transactions on Image Processing*, Vol. 10, pp. 1–13, January 2001.
68. Cheung, K.-W., C.-H. Cheung, and W.-L. Po, 1999. A novel multiwavelet-based integer transform for lossless image coding. In *International Conference on Image Processing*, Vol. 1, pp. 444–447.
69. Grangetto, M., E. Magli, and G. Olmo, 2000. Minimally non-linear integer wavelets for image coding. In *International Conference on Accoustics, Speech and Signal Processing*, Vol. 4, pp. 2039–2042.

70. Marpe, D., G. Blätermann, J. Ricke, and P. Maas, 2000. A two-layered wavelet-based algorithm for efficient lossless and lossy image compression. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 10, pp. 1094–1102, October 2000.
71. Egger, O., and M. Kunt, 1995. Embedded zero tree based lossless image coding. In *International Conference on Image Processing*, Washington, DC, Vol. 3, pp. 616–619.
72. Said, A., and W. Pearlman, 1996. An image multiresolution representation for lossless and lossy compression. *IEEE Transactions on Image Processing*, Vol. 5, pp. 1303–1310, September 1996.
73. Creusere, C., 1998. Successive coefficient refinement for embedded lossless image compression. In *International Conference on Image Processing*, Vol. 1, pp. 521–525.
74. Ramaswamy, V., N. Ranganathan, and K. Namuduri, 1999. Performance analysis of wavelets in emebded zerotree-based lossless image coding schemes. *IEEE Transactions on Signal Processing*, Vol. 47, pp. 884–889, March 1999.

Text Compression

AMAR MUKHERJEE
FAUZIA AWAN¹

10.1 INTRODUCTION

In recent times, we have seen an unprecedented explosion of textual information through the use of the Internet, digital libraries, and information retrieval systems. The advent of office automation systems and newspaper, journal, and magazine repositories has brought the issue of maintaining archival storage for search and retrieval to the forefront of research. As an example, the TREC [33] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. Text compression is concerned with techniques for representing the digital text data in alternate representations that take less space. It not only helps conserve the storage space for archival and online data, it also helps system performance by requiring less secondary storage (disk or CD Rom) access and improves network transmission bandwidth utilization by reducing the transmission time.

Data compression methods are generally classified as *lossless* or *lossy*. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, fingerprint data, astronomical images, and databases containing mostly vital numerical data, tables, and text information. In contrast, lossy compression schemes allow some deterioration and are generally used for video, audio, and still-image applications. The deterioration of the quality of lossy images is usually not detectable by the human perceptual system, and the compression systems exploit this by a process called “*quantization*” to achieve compression by a factor of 10 to a factor of a couple of hundreds. Many lossy algorithms use lossless methods at the final stage of the

¹ The opinions expressed are those of the author and not those of Microsoft.

encoding, underscoring the importance of lossless methods for both lossy and lossless compression applications. This chapter will be concerned with lossless algorithms for textual information.

We will first review in Section 10.2 the basic concepts of information theory applicable in the context of lossless text compression. We will then briefly describe the well-known compression algorithms in Section 10.3. A detailed description of these and other methods is now available in several excellent recent books [16, 28, 29, 35, and others]. In Sections 10.4 and 10.5, we present our own research on text compression. In particular, we present a number of preprocessing techniques that transform the text in some intermediate forms that produce better compression performance. We give test results of compression and timing performance with respect to text files in three corpora: Canterbury, Calgary [8], and Gutenberg [17] corpora. We conclude our chapter with a discussion of the compression utility web site that is now integrated with the Canterbury web site for availability via the Internet.

10.2 INFORMATION THEORY BACKGROUND

The general approach to text compression is to find a representation of the text requiring fewer binary digits. In its uncompressed form each character in the text is represented by an 8-bit ASCII code. It is common knowledge that such a representation is not very efficient because it treats frequent and less frequent characters equally. It makes intuitive sense to encode frequent characters with a smaller number of bits (less than 8) and less frequent characters with a larger number of bits (possibly more than 8 bits) in order to reduce the *average number of bits per character* (BPC). In fact this principle was the basis of the invention of the so-called Morse code and the famous Huffman code developed in the early 1950s. Huffman code typically reduces the size of the text file by about 50–60% or provides a compression rate of 4–5 BPC [35] based on the statistics of frequency of characters. In the late 1940s, Claude E. Shannon laid the foundation of information theory and modeled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [31, 32]. The information content of the text source can then be quantified by the entity called *entropy H* given by

$$H = - \sum_i p_i \log p_i, \quad (10.1)$$

where p_i denotes the probability of occurrence of the i th symbol of the alphabet in the text, the sum of all symbol probabilities is unity, the logarithm is with respect to base 2, and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the i th symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the *average number of BPC* required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called *static* probability, it is called an Order (0) model. Under this model, it is also possible to compute the *dynamic* probabilities, which can be roughly described as follows. At the beginning when no text symbol has emerged out of the source, assume that every symbol is equiprobable. As new symbols of the text emerge from the source, revise the probability values according to the actual frequency distribution of the symbols at that time. In general, an Order (k) model can be defined where the probabilities are computed based on the probability of distribution of the $(k + 1)$ -grams of symbols or equivalently, by taking into account the context of the preceding k symbols. A value

of $k = -1$ is allowed and is reserved for the situation when all symbols are considered equiprobable; that is, $p_i = 1/|A|$, where $|A|$ is the size of the alphabet A . When $k = 1$, the probabilities are based on *bigram* statistics or equivalently on the context of just one preceding symbol and similarly for higher values of k . For each value of k , there are two possibilities: the static model and the dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus*, which is a collection of text samples representing a particular domain of application (viz., English literature, physical sciences, life sciences). If one is interested in a more precise static model for a given text, a *semistatic* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to *prime* or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of the model, there is entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [31]. The "worst" model from an information theoretic point of view is the Order (-1) model, the equiprobable model, giving the maximum value H_m of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference between the maximum entropy H_m and the actual entropy H . As we build better and better models by going to higher order k , the value of entropy will be lower, yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible entropy and exploiting a maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again readable), speed of execution of compression and decompression algorithms, and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means a reduced bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* that may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless; that is, it does not introduce error during transmission and it has a *channel capacity* C that is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [31]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on average more than C/H symbols per second [31]. In other words, to use the available bandwidth effectively, H should be as low as possible, which means employing a compression scheme that yields minimum BPC.

10.3 CLASSIFICATION OF LOSSLESS COMPRESSION ALGORITHMS

The lossless algorithms can be classified into three broad categories: *statistical methods*, *dictionary methods*, and *transform-based methods*. We will give a very brief review of these methods in this section.

10.3.1 Statistical Methods

The classic method of statistical coding is *Huffman* coding [19]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of

the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol that replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels “0” and “1” in any arbitrary order. The process is then repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0’s and 1’s in the path from the root to a leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree must be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that must be transmitted once from the sender to receiver and this constitutes an initial overhead of the algorithm. *This overhead is usually ignored in publishing the BPC results for Huffman code in the literature.* The Huffman codes for all the symbols have what is called the *prefix property*, which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable* (UD). This allows forming a code for a sequence of symbols by concatenating just the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code and may not obey the Morse principle and that a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, is prefix, and is also optimum within 1 bit of the entropy H . Huffman code is indeed optimum if the probabilities are $1/2^k$, where k is a positive integer. There are also Huffman codes called *canonical* Huffman codes, which use a lookup table or dictionary rather than a binary tree for fast encoding and decoding [28, 35].

Note that in the construction of the Huffman code, we started with a model. The efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller, resulting in a shorter average code length. As an example, a word-based Huffman code is constructed by collecting the statistics of words in the text and building a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English word dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes take longer for both encoding and decoding because the Huffman tree must be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable-length code because a message of a fixed length may have variable-length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable-size message into fixed-length binary sequence. Arithmetic code is inherently adaptive, does not use any lookup table or dictionary, and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: At the beginning the semiclosed interval $[0, 1]$ is partitioned into $|A|$ equal sized semiclosed intervals under the equiprobability assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 , of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol, a_2 , in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ subintervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new subinterval for a_2 . The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbols. A simpler version of this algorithm

could use the same static distribution of probability at each iteration, avoiding recomputation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [28, 29, 35] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coder*. These methods normally use an Order (0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently. One of the best known modelers is “*prediction by partial match*” (PPM) [10, 23]. PPM uses a finite context Order (k) model, where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a table or a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level and for each distinct context at that level (except the level with $k = -1$), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct contexts encountered at that context level for the purpose of calculating its probability. The escape character is required to handle the situation when the encoder encounters a new context never encountered before at any context level to give the decoder a signal that the context length must be reduced by 1. During the encoding process, the algorithm estimates the probability of the occurrence of some given *next character* in the text stream as follows: The algorithm tries to find the current context of maximum length k for this character in the context table or trie. If the context is not found, it passes the probability of the escape character at this level for this context and goes down one level to the $k - 1$ context table to find the current context of length $k - 1$ and the process is repeated. If it continues to fail to find the context, it may go down ultimately to the $k = -1$ level corresponding to the equiprobable level for which the probability of any next character is $1/|A|$. If, on the other hand, a context of length q , $0 \leq q \leq k$, is found, then the probability of this next character is estimated to be the product of probabilities of escape characters at levels $k, k - 1, \dots, q + 1$ multiplied by the probability of the context found at the q th level.

This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note that at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few contexts have been seen. As the encoding proceeds, longer and longer contexts become available. The method to assign probability to the escape character is called method C and is as follows: At any level, with the current context, let the total number of symbols seen previously be n_t and let n_d be the total number of *distinct* context. Then the probability of the escape character is given by $n_d/(n_d + n_t)$. Any character other than that which appeared in this context n_c times will have a probability $n_c/(n_d + n_t)$. The intuitive explanation of this method, based on experimental evidence, is that if many distinct contexts are encountered, then the escape character will have higher probability, but if these distinct contexts tend to appear too many times, then the probability of the escape character decreases. The PPM method using method C for probability estimation is called the PPMC algorithm. There are a few other variations: PPMA uses method A, which simply assigns a count of 1 to the escape character, yielding its probability to be $1/(n_t + 1)$ and the probability of the character which appeared n_c times is $n_c/(n_t + 1)$. PPMB is very similar to PPMC except that the probability of a symbol is $(n_c - 1)/(n_d + n_t)$; that is, 1 is subtracted from the count n_c . If n_c is 1, since $n_c - 1$ becomes 0, no probability is assigned to the symbol and the count for the escape character is increased by 1. In PPMD, the escape character gets a probability of $n_d/2n_t$ and the symbol gets a probability $(2n_c - 1)/2n_t$. All of these methods have been proposed based on practical experience and have only some intuitive explanation but no theoretical basis. PPMD performs better than PPMC, which is better than either PPMA or PPMB. In one version of PPM, called PPM*, an arbitrary-length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as well as PPM* [11]. Although the PPM family of algorithms performs better than other compression algorithms in

terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) is another modeling scheme that is equivalent to the finite context model but uses a finite-state machine to estimate the probabilities of the input symbols which are bits rather than bytes or symbols as in PPM [12]. The model is adaptive and starts with a single state fsa (finite-state automaton) with transitions into itself. In general, the transitions from a given state are marked by $0/p$ or $1/q$, where p and q denote non-zero counts of the number of transitions from the given state to two other states (which may include the given state), respectively. Initially, the values of p and q are set to 1 to handle the zero-frequency problem. The estimate of probability of a 0 being the next input is $p/(p + q)$ and 1 being the next input is $q/(p + q)$. These probability values are used by the entropy coder (such as an arithmetic coder) to estimate the interval in the cumulative probability space for the input. If the next symbol is 0, it will then change the count p to $p + 1$, make the appropriate state change, and continue. Similarly, for a 1 input it will change the value of q to $q + 1$. The machine adapts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state s is used heavily for input transitions from another state t (caused by either 1 or 0 input), s is *cloned* into two states by introducing a new state s_c in which some of the transitions from state t are directed. The threshold value of either p or q at which cloning must be triggered is agreed upon by the encoder and the decoder ahead of time. The ratio p/q of output transitions from the original state s is kept the same for the cloned state s_c , but the count values p and q from t to s and from t to s_c are adjusted for both the cloned state and the original state so that their total sum does not change. This is necessary to preserve the total number of transitions encountered by the fsa up to the time prior to cloning. The decoder performs the reverse starting with a single-state fsa and mimicking the cloning operation of the encoder whenever necessary. For further details, refer to [29, 35].

The bit-wise encoding and decoding take longer and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent [5].

10.3.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary* or *codebook* of words or text strings previously encountered in the text input, and data compression is achieved by replacing strings in the text with a reference to the string in the dictionary. The dictionary is *dynamic* or *adaptive* in the sense that it is constructed by adding new strings that are being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary, like the word dictionary, to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv–Lempel or LZ77 coding [37] in which the text prior to the current symbol constitutes the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if so, they are replaced by a reference giving its relative starting position in the text. Because of the pattern-matching operation the encoding takes longer but the process has been fine-tuned with the use of hashing techniques and special data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding, includes one extra character in a previously coded string in the encoding scheme. A more popular variant of the LZ78 family is the so-called LZW algorithm, which led to the widely used *compress* algorithm. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the

existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created, appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and is made available to future steps. Many other variants of the LZ77 and LZ78 compression family have been reported in the literature (see [28, 29] for further references).

10.3.3 Transform-Based Methods: The Burrows–Wheeler Transform (BWT)

The word “transform” has been used to describe this method because the text undergoes a transformation, which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrows–Wheeler transform [6] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix are the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as the *block sorting* transform [14]. The repetition of the same character in the block might slow the sorting process; to avoid this, a run-length encoding step could precede the transform step. The Bzip2 compression algorithm based on the BWT transform uses this step and other steps as follows: The output of the BWT transform stage undergoes a final transformation using either move-to-front (MTF) encoding or distance coding (DC) [1], which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz., 0, 1, or 2) out of possible maximum value of $|A|$. This sequence of numbers is then sent to an entropy coder (Huffman or arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then the inverse of MTF or DC, and then an inverse of BWT. The inverse of BWT obtains the original text given (L, r) . This is done easily by noting that the first column of M , denoted F , is simply a sorted version of L . Define an index vector Tr of size $|L|$ such that $Tr[j] = i$ if and only if both $L[j]$ and $F[i]$ denote the k th occurrence of a symbol from A . Since the rows of M are cyclic rotations of the text, the elements of L precede the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text, which leads to a simple algorithm to reconstruct the original text.

10.3.4 Comparison of Performance of Compression Algorithms

An excellent discussion of the performance comparison of the important compression algorithms can be found in [35]. In general, the performance of compression methods depends on the type of data being compressed and there is a trade-off between compression performance, speed, and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as follows: order zero arithmetic; order zero Huffman giving over 4 BPC; the LZ family of algorithms, whose performance ranges from 4 BPC to around 2.5 BPC (gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word-based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are bzip2 (based on BWT), DMC, and PPM, all giving BPC ranging from 2.1 to 2.4. PPM is theoretically the best but is extremely

slow as is DMC; bzip2 strikes a middle ground—it gives better results than Gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (Gzip) are fastest for decompression, then the LZ78 technique, and then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications, whereas arithmetic coding is preferable in adaptive and on-line coding. Bzip2 decodes faster than most other methods and it achieves good compression as well. A lot of new research on Bzip2 (see Section 10.4) has been carried out recently to push the performance envelope of Bzip2 in terms of both compression ratio and speed, and as a result bzip2 has become a strong contender to replace the popularity of Gzip and compress.

New research is under way to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation regarding lowering the compression ratio. To get a significant further improvement in compression, other means such as transforming the text before actual compression and the use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of the English language and estimated that the English language has an entropy of around 1.3 BPC. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio. Of course, combining compression performance with other performance metrics like speed, memory overhead, and on-line capabilities seems to pose even a bigger challenge.

10.4 TRANSFORM-BASED METHODS: STAR (*) TRANSFORM AND LENGTH-INDEX PRESERVING TRANSFORM

In this section we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called the Star Transform (also denoted *-encoding). We then present four new transforms called length-index preserving transform (LIPT), initial letter preserving transform (ILPT), number index transform (NIT), and letter index transform (LIT), which produce better results.

The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. The typical size of a dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, and decompression times) of our proposed preprocessing techniques using three corpora: the Calgary, Canterbury, and Gutenberg corpora.

10.4.1 Star (*) Transformation

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in a word with a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [15, 20, 21]. For an English language dictionary D of size 60,000 words, we observed that we needed at most two characters of the original words

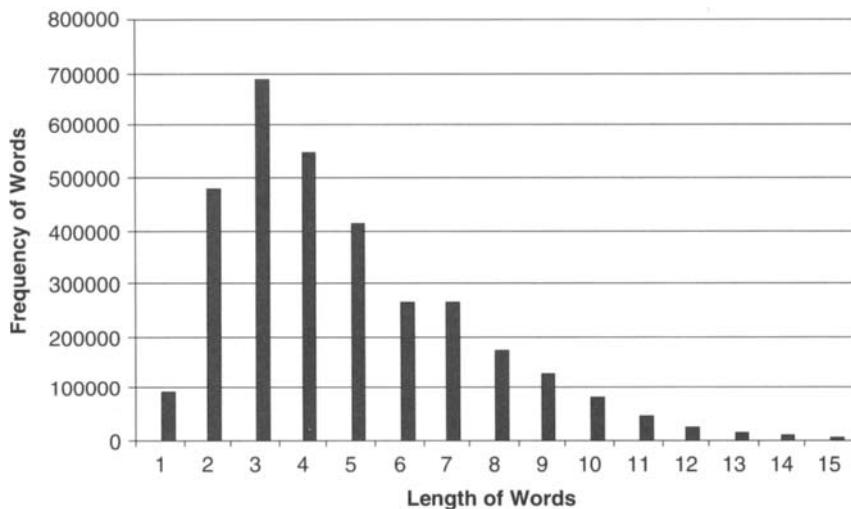
to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as a unique representation can be defined. The dictionary is divided into subdictionaries D_s containing words of length, $1 \leq s \leq 22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1, viz., “a” and “I”.

The following encoding scheme is used for the words in D_s : The first word is represented as sequence of s stars. The next 52 words are represented by a sequence of $s - 1$ stars followed by a single letter from the alphabet: $= (a, b, \dots, z, A, B, \dots, Z)$. The next 52 words have a similar encoding except that the single letter appears in the next to last position. This will continue until all the letters occupy the first position in the sequence. The following group of words has $s - 2$ *'s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53^s unique encodings, which is more than sufficient for English words. A large fraction of these combinations are never used; for example, for $s = 2$, there are only 17 words and for $s = 8$, there are about 9000 words in the English dictionary. As an example of star encoding the sentence "*Our philosophy of compression is to transform the text into some intermediate form which can be compressed with better efficiency and which exploits the natural redundancy of the language in making this transformation*" can be *-encoded as

There are exactly five 2-letter words in the sentence (of, is, to, be, in) which can be uniquely encoded as (**, *a, *b, *c, *d) and the other groups of words can be uniquely encoded in a similar manner. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text, making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz., a new word in the lexicon), it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks, and capitalization. The space character is used as word separator. The character “~” at the end of an encoded word indicates that the first letter of the input text word is capitalized. The character “^” indicates that all the characters in the input word are capitalized. A capitalization mask, preceded by the character “^”, is placed at the end of an encoded word to denote capitalization of characters other than the first letter and all capital letters. The character “\” is used as an escape character for encoding the occurrences of *, ~, ` , ^, and \ in the input text. The transformed text can now be the input to any available lossless text compression algorithm, including Bzip2, where the text undergoes two transformation: first the *-transform and then a BWT transform.

10.4.2 Length-Index Preserving Transform (LIPT)

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in the English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k -grams, and words in the form of probability models, of course, forms the cornerstone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider the length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [8], and Gutenberg

**FIGURE 10.1**

Frequency of English words versus length of words in the test corpus.

corpora [17]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Fig. 10.1.

It can be seen that most words lie in the range of length 1 to 10. Most words have length 2 to 5. The word length and word frequency results provided a basis to build context in the transformed text. LIPT can be regarded as the first step of a multistep compression algorithm such as Bzip2, which includes run-length encoding, BWT, move-to-front encoding, and Huffman coding. LIPT can be used as an additional component in Bzip2 before run-length encoding or it can simply replace it. Compared to the $*$ -transform, we also made a couple of modifications to improve the timing performance of LIPT. For $*$ -transform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time, which degrades the execution times. The situation can be improved by presorting the words lexicographically and doing a binary search on the sorted dictionary during both the compression and decompression stages. The other new idea that we introduce is to be able to access the words during the decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the address of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce an artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. The LIPT encoding scheme makes use of the recurrence of same-length words in the English language to create context in the transformed text that the entropy coders can exploit.

LIPT uses a static English language dictionary of 59,951 words, having a size of around 0.5 MB. LIPT uses a transform dictionary of around 0.3 MB. There is one-to-one mapping of a word from the English to the transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done off-line), we need the source English dictionary to be sorted on blocks of lengths, and words in each block should be sorted according to their frequency of use.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i , each containing words of length i , where $i = 1, 2, \dots, n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for

all words in each dictionary D_i . $D_i[j]$ denotes the j th word in the dictionary D_i . In LIPT, the word $D_i[j]$ in the dictionary D is transformed as $*c_{len}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT} , where c_{len} stands for a letter in the alphabet [a–z, A–Z] each denoting a corresponding length [1–26, 27–52] and each c is in [a–z, A–Z]. If $j = 0$, then the encoding is $*c_{len}$. For $j > 0$, the encoding is $*c_{len} c[c][c]$. Thus, for $1 \leq j \leq 52$ the encoding is $*c_{len}c$; for $53 \leq j \leq 2756$ it is $*c_{len}cc$, and for $2757 \leq j \leq 140,608$ it is $*c_{len}ccc$. Thus, the 0th word of length 10 in the dictionary D will be encoded as “*j” in D_{LIPT} , $D_{10}[1]$ as “*ja”, $D_{10}[27]$ as “*jaA”, $D_{10}[53]$ as “*jaa”, $D_{10}[79]$ as “*jaA”, $D_{10}[105]$ as “*jba”, $D_{10}[2757]$ as “*jaaa”, $D_{10}[2809]$ as “*jaba”, and so on.

The transform must also be able to handle special characters, punctuation marks, and capitalization. The character “*” is used to denote the beginning of an encoded word. The handling of capitalization and special characters is the same as in *-encoding. Our scheme allows for a total of 140,608 encodings for each word length. Since the maximum length of English words is around 22 and the maximum number of words in any D_i in our English dictionary is less than 10,000, our scheme covers all English words in our dictionary and leaves enough room for future expansion. If the word in the input text is not in the English dictionary (viz., a new word in the lexicon), it will be passed to the transformed text unaltered.

The decoding steps are as follows: The received encoded text is first decoded using the same backend compressor used at the sending end and the transformed LIPT text is recovered. The words with “*” represent transformed words and those without “*” represent non-transformed words and do not need any reverse transformation. The length character in the transformed words gives the length block and the next three characters give the offset in the respective block. The words are looked up in the original dictionary D in the respective length block and at the respective position in that block as given by the offset characters. The transformed words are replaced with the respective words from dictionary D . The capitalization mask is then applied.

10.4.3 Experimental Results

The performance of LIPT is measured using Bzip2 -9 [6, 9, 22, 30], PPMD (order 5) [11, 23, 28], and Gzip -9 [28, 35] as the backend algorithms in terms of average BPC. Note that these results include some amount of precompression because the size of the LIPT text is smaller than the size of the original text file. By average BPC we mean the unweighted average (simply taking the average of the BPC of all files) over the entire text corpus. The test corpus is shown in Table 10.1. Note that all the files given in Table 10.1 are text files extracted from the corpora.

We used SunOS Ultra-5 to run all our programs and to obtain results. We are using a 60,000-word English dictionary that takes 557,537 bytes. The LIPT dictionary takes only 330,636 bytes compared to the *-encoded dictionary, which takes as much storage as that of the original dictionary. LIPT achieves a bit of compression in addition to preprocessing the text before application to any compressor.

The results can be summarized as follows: The average BPC using the original Bzip2 is 2.28, and using Bzip2 with LIPT gives an average BPC of 2.16, a 5.24% improvement. The average BPC using the original PPMD (order 5) is 2.14, and using PPMD with LIPT gives an average BPC of 2.04, an overall improvement of 4.46%. The average BPC using the original Gzip -9 is 2.71, and using Gzip -9 with LIPT the average BPC is 2.52, a 6.78% improvement.

Figure 10.2 gives a bar chart comparing the BPC of the original Bzip2, PPMD, and the compressors in conjunction with LIPT for a few text files extracted from our test corpus. From Fig. 10.2 it can be seen that Bzip2 with LIPT (the second bar in Fig. 10.2) is close to the original

Table 10.1 Text Files and Their Sizes (a) from Calgary Corpus and (b) from Canterbury Corpus and Project Gutenberg Corpus That Were Used in Our Tests

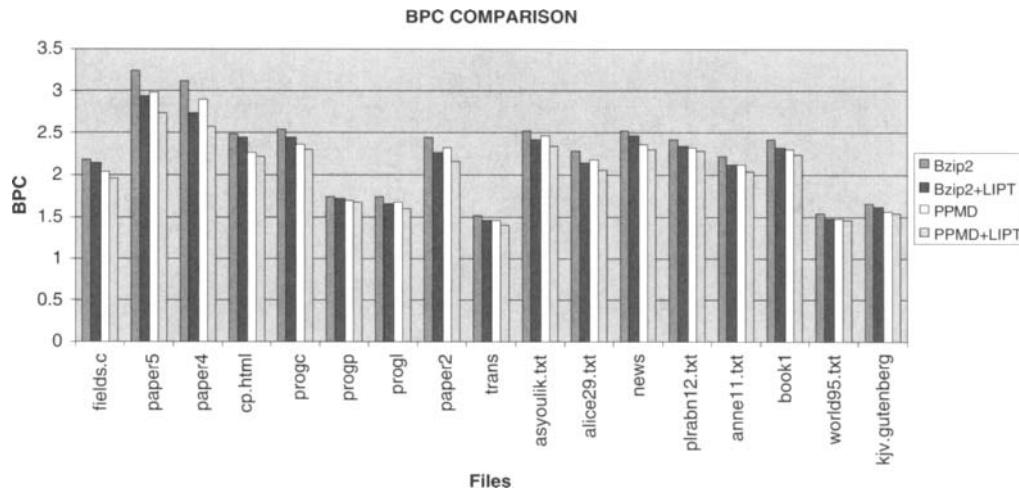
File Name	Actual Size
(a)	
Calgary	
Bib	111,261
book1	768,771
book2	610,856
News	377,109
paper1	53,161
paper2	82,199
paper3	46,526
paper4	13,286
paper5	11,954
paper6	38,105
ProgC	39,611
ProgI	71,646
ProgP	49,379
Trans	93,695
(b)	
Canterbury	
alice29.txt	152,089
asyoulik.txt	125,179
cp.html	24,603
fields.c	11,150
grammar.lsp	3,721
lcet10.txt	426,754
plrabn12.txt	481,861
xargs.1	4,227
bible.txt	4,047,392
kjv.Gutenberg	4,846,137
world192.txt	2,473,400
Project Gutenberg	
1musk10.txt	1,344,739
anne11.txt	586,960
world95.txt	2,988,578

PPMD (the third bar in Fig. 10.2) in bits per character. From Fig. 10.2 it can also be seen that in instances like paper5, paper4, progI, paper2, asyoulik.txt, and alice29.txt, Bzip2 with LIPT beats the original PPMD in terms of BPC. The difference between the average BPC for Bzip2 with LIPT (2.16) and that for the original PPMD (2.1384) is only around 0.02 bits; i.e., the average BPC for Bzip2 with LIPT is only around 1% more than that for the original PPMD. Table 10.2 gives a summary comparison of BPC for the original Bzip2 -9, PPMD (order 5), Gzip -9, Huffman (character based), word-based arithmetic coding, and these compressors with Star-encoding and LIPT. The data in Table 10.2 show that LIPT performs much better than Star-encoding and original algorithms except for character-based Huffman and Gzip -9.

Table 10.2 also shows that Star-encoding gives a better average BPC performance for character-based Huffman, Gzip, and Bzip2 but gives a worse average BPC performance for word-based

Table 10.2 Summary of BPC Results

	Original (BPC)	*-encoded (BPC)	LIPT (BPC)
Huffman (character based)	4.87	4.12	4.49
Arithmetic (word based)	2.71	2.90	2.61
Gzip-9	2.70	2.52	2.52
Bzip2	2.28	2.24	2.16
PPMD	2.13	2.13	2.04

**FIGURE 10.2**

Bar chart giving comparison of Bzip2, Bzip2 with LIPT, PPMD, and PPPMD with LIPT.

arithmetic coding and PPMD. This is due to the presence of the non-English words and special symbols in the text. For a pure text file, for example, the dictionary itself, the star dictionary has a BPC of 1.88 and the original BPC is 2.63 for PPMD. The improvement is 28.5% in this case. Although the average BPC for Star-encoding is worse than that for the original PPMD, there are 16 files that show improved BPC and 12 files that show worse BPC. The number of words in the input text that are also found in English dictionary D is an important factor for the final compression ratio. For character-based Huffman, Star-encoding performs better than the original Huffman and LIPT with Huffman. This is because in Star-encoding there are repeated occurrences of the character “*”, which gets the highest frequency in the Huffman codebook and is thus encoded with the lowest number of bits, resulting in better compression results than for the original and the LIPT files.

We focus our attention on LIPT over Bzip2 (which uses BWT), Gzip, and PPM algorithms because Bzip2 and PPM outperform other compression methods and Gzip is commercially available and commonly used. Of these, a BWT-based approach has proved to be the most efficient and a number of efforts have been made to improve its efficiency. The latest efforts include works by Balkenhol *et al.* [4], Seward [30], Chapin [9], and Arnavut [1]. PPM, on the other hand, gives a better compression ratio than BWT but is very slow in terms of execution time. A number of efforts have been made to reduce the time for PPM and also to improve the compression ratio.

Table 10.3 BPC Comparison of Approaches Based on BWT

File [Reference]	Average BPC
MBSWIC [1]	2.21
BKS98 [4]	2.105
Best x of $2x - 1$ [9]	2.11
Bzip2 with LIPT	2.07

Table 10.4 BPC Comparison of New Approaches Based on Prediction Models

File [Reference]	Average BPC
Multialphabet CTW order 16 [27]	2.021
NEW Effros [13]	2.026
PPMD (order 5) with LIPT	1.98

Sadakane *et al.* [27] have given a method where they have combined PPM and CTW [35] to get better compression. Effros [13] has given a new implementation of PPM* with the complexity of BWT. Tables 10.3 and 10.4 give a comparison of compression performance of our proposed transform, which shows that LIPT has a better average BPC than all the other methods cited.

The Huffman compression method also needs the same static dictionary to be shared at both the sender and the receiver ends, as does our method. The canonical Huffman [35] method assigns variable-length addresses to words using bits, and LIPT assigns variable-length offset in each length block using letters of the alphabet. Due to these similarities we compare the word-based Huffman with LIPT (we used Bzip2 as the compressor). Huffman and LIPT both sort the dictionary according to frequency of use of words. Canonical Huffman assigns a variable address to the input word, building a tree of locations of words in the dictionary and assigning 0 or 1 to each branch of the path. LIPT exploits the structural information of the input text by including the length of the word in encoding. LIPT also achieves a precompression due to the variable offset scheme. In Huffman, if new words are added, the whole frequency distribution must be recomputed as do the Huffman codes for them. Comparing the average BPC, the Managing Gigabyte [35] word-based Huffman model has 2.506 BPC for our test corpus. LIPT with Bzip2 has a BPC value of 2.17. The gain is 13.44%. LIPT does not give an improvement over word-based Huffman for files with mixed text such as source files for programming languages. For files with more English words, LIPT shows a consistent gain.

LIPT gives better results with some of the newer compression methods reported in the literature and web sites such as YBS [36], RK [25, 26], and PPMonstr [24]. The average BPC using LIPT along with these methods is around 2.00, which is better than any of these algorithms as well as the original Bzip2 and PPMD. For details, the reader is referred to [2, 3].

10.4.4 Timing Performance Measurements

The improved compression performance of our proposed transform comes with a penalty of degraded timing performance. For off-line and archival storage applications, such penalties in timing are quite acceptable if a substantial savings in storage space can be achieved. The increased

compression/decompression times are due to frequent access to a dictionary and its transform dictionary. To alleviate the situation, we have developed efficient data structures to expedite access to the dictionaries and memory management techniques using caching. Realizing that certain on-line algorithms might prefer not to use a preassigned dictionary, we also have been working on a new family of algorithms, called M5zip, to obtain the transforms dynamically with no dictionary and with small dictionaries (7947 words and 10,000 words), which will be reported in future papers.

The experiments were carried out on a Sun Microsystems Ultra Sparc-IIi 360-MHz machine housing SunOS 5.7. In our experiments we compare compression times of Bzip2, Gzip, and PPMD against Bzip2 with LIPT, Gzip with LIPT, and PPMD with LIPT. During the experiments we used the -9 option for Gzip. This option supports better compression. Average compression time, for our test corpus, using LIPT with Bzip2 -9, Gzip -9, and PPMD is 1.79 times slower, 3.23 times slower, and fractionally (1.01 times) faster than original Bzip2, Gzip, and PPMD, respectively. The corresponding results for decompression times are 2.31 times slower, 6.56 times slower, and almost the same compared to original Bzip2, Gzip, and PPMD, respectively. Compression using Bzip2 with LIPT is 1.92 times faster and decompression is 1.98 times faster than original PPMD (order 5). The increase in time over standard methods is due to time spent in preprocessing the input file. Gzip uses the -9 option to achieve maximum compression; therefore we find that the times for compression using Bzip2 are less than those for Gzip. When maximum compression option is not used, Gzip runs much faster than Bzip2.

Decompression time for methods using LIPT includes decompression using compression techniques plus reverse transformation time. Bzip2 with LIPT decompresses 2.31 times slower than original Bzip2, Gzip with LIPT decompresses 6.56 times slower than Gzip, and PPMD with LIPT is almost the same.

10.5 THREE NEW TRANSFORMS—ILPT, NIT, AND LIT

We will briefly describe our three new lossless reversible text transforms based on LIPT. We give experimental results for the new transforms and discuss them briefly. For details on these transformations and experimental results the reader is referred to [3]. Note that there is no significant effect on the time performance as the dictionary loading method remains the same and the number of words also remains the same in the static English dictionary D and transform dictionaries. Hence we will give only the BPC results obtained with different approaches for the corpus.

The *Initial Letter Preserving transform* is similar to LIPT except that we sort the dictionary into blocks based on the lexicographic order of starting letter of the words. We sort the words in each letter block according to descending order of frequency of use. The character denoting length in LIPT (character after “*”) is replaced by the starting letter of the input word; i.e., instead of $*c_{len}[c][c][c]$, for ILPT, it becomes $*c_{init}[c][c][c]$, where c_{init} denotes the initial (staring) letter of the word. Everything else is handled the same way as in LIPT. Bzip2 with ILPT has an average BPC of 2.12 for all the files in all the corpora combined. This means that Bzip2 -9 with ILPT shows 6.83% improvement over the original Bzip2 -9. Bzip2 -9 with ILPT shows 1.68% improvement over Bzip2 with LIPT.

The *Number Index Transform* scheme uses variable addresses based on letters of the alphabet instead of numbers. We wanted to compare this using a simple linear addressing scheme with numbers, i.e., giving addresses 0–59,950 to the 59,951 words in our dictionary. Using this scheme on our English dictionary D , sorted according to length of words and then sorted according to frequency within each length block, gave deteriorated performance compared to LIPT. So we sorted the dictionary globally according to descending order of word usage frequency. No blocking

was used in the new frequency-sorted dictionary. The transformed words are still denoted by character “*”. The first word in the dictionary is encoded as “*0”, the 1000th word is encoded as “*999”, and so on. Special character handling is same as in LIPT. We compare the BPC results for Bzip2 -9 with the new transform NIT with Bzip2 -9 with LIPT. Bzip2 with NIT has an average BPC of 2.12 for all the files in all the corpora combined. This means that Bzip2 -9 with NIT shows 6.83% improvement over the original Bzip2 -9. Bzip2 -9 with NIT shows 1.68% improvement over Bzip2 with LIPT.

Combining the approach taken in NIT and using letters to denote offset, we arrive at another transform which is same as NIT except that now we use letters of the alphabet [a-z; A-Z] to denote the index or the linear address of the words in the dictionary, instead of numbers. This scheme is called the *Letter Index Transform*. Bzip2 with LIT has an average BPC of 2.11 for all the files in all corpora combined. This means that Bzip2 -9 with LIT shows 7.47% improvement over the original Bzip2 -9. Bzip2 -9 with LIT shows 2.36% improvement over Bzip2 with LIPT.

The transform dictionary sizes vary with the transform. The original dictionary takes 557,537 bytes. The transformed dictionaries for LIPT, ILPT, NIT, and LIT are 330,636, 311,927, 408,547, and 296,947 bytes, respectively. Note that LIT has the smallest size dictionary and shows uniform compression improvement over other transforms for almost all the files.

Because of its performance is better than that of other transforms, we compared LIT with PPMD, YBS, RK, and PPMonstr. PPMD (order 5) with LIT has an average BPC of 1.99 for all the files in all corpora combined. This means that PPMD (order 5) with LIT shows 6.88% improvement over the original PPMD (order 5). PPMD with LIT shows 2.53% improvement over PPMD with LIPT. RK with LIT has an average BPC value lower than that of RK with LIPT. LIT performs well with YBS, RK, and PPMonstr, giving 7.47, 5.84, and 7.0% improvement, respectively, over the original methods. It is also important to note that LIT with YBS outperforms Bzip2 by 10.7% and LIT with PPMonstr outperforms PPMD by 10%.

LIPT introduces frequent occurrences of common characters for BWT and good context for PPM also as it compresses the original text. Cleary *et al.* [11] and Larsson [22] have discussed the similarity between PPM and Bzip2. PPM uses a probabilistic model based on the context depth and uses the context information explicitly. On the other hand, the frequency of similar patterns and the local context affect the performance of BWT implicitly. Fenwick [14] also explains how BWT exploits the structure in the input text. LIPT introduces added structure along with smaller file size, leading to better compression after Bzip2 or PPMD is applied. LIPT exploits the distribution of words in the English language based on the length of the words as given in Fig. 10.1. The sequence of letters to denote the address also has some inherent context depending on how many words are in a single group, which also opens another opportunity to be exploited by the backend algorithm at the entropy level.

There are repeated occurrences of words with the same length in a usual text file. This factor contributes to introducing good and frequent context and thus higher probability of occurrence of same characters (space, “*”, and characters denoting length of words) that enhance the performance of Bzip2 (which uses BWT) and PPM as proved by results given earlier in this report. LIPT generates an encoded file, which is smaller in size than the original text file. Because of the small input file along with a set of artificial but well-defined deterministic contexts, both BWT and PPM can exploit the context information very effectively, producing a compressed file that is smaller than the file without using LIPT.

Our research has shown that the compression given by compression factor C is inversely proportional to the product of file size reduction factor F achieved by a transform and entropy S_t . Smaller F and proportionally smaller entropy of the transformed file mean higher compression. For theoretical details, the reader is referred to [3].

Our transforms keep the word level context of the original text file but adopt a new context structure at the character level. The frequency of the repeated words remains the same in both the

original and the transformed files. The frequency of characters is different. Both of these factors along with the reduction in file size contribute toward the better compression achieved with our transforms. The context structure affects the entropy S or S_t and reduction in file size affects F . We have already discussed the finding that compression is inversely proportional to the product of these two variables. In all our transforms, to generate our transformed dictionary, we have sorted the words according to frequency of usage in our English dictionary D . For LIPT, words in each length block in English dictionary D are sorted in descending order according to frequency. For ILPT, there is a sorting based on descending order of frequency inside each initial letter block of English dictionary D . For NIT, there is no blocking of words. The whole dictionary is one block. The words in the dictionary are sorted in descending order of frequency. LIT uses the same structure of dictionary as NIT. Sorting of words according to frequency plays a vital role in the size of the transformed file and also its entropy. Arranging the words in descending order of usage frequency results in shorter codes for more frequently occurring words and longer codes for less frequently occurring words. This fact leads to smaller file sizes.

10.6 CONCLUSIONS

In this chapter, we have given an overview of classical and recent lossless text compression algorithms and then presented our research on text compression based on transformation of text as a preprocessing step for use by the available compression algorithms. For comparison purposes, we were primarily concerned with Gzip, Bzip2, a version of PPM called PPMD, and word-based Huffman. We gave a theoretical explanation of why our transforms improved the compression performance of the algorithms. We have developed a web site (<http://vlsi.cs.ucf.edu>) as a test bed for all compression algorithms. To use this, one must go to the “on-line compression utility” and the client could then submit any text file for compression using all the classical compression algorithms, some of the most recent algorithms including Bzip2, PPMD, YBS, RK, and PPMonstr, and, of course, all the transform-based algorithms that we developed and reported in this chapter. The site is still under construction and is evolving. One nice feature is that the client can submit a text file and obtain statistics of all compression algorithms presented in the form of tables and bar charts. The site is being integrated with the Canterbury web site.

ACKNOWLEDGMENTS

The research reported in this chapter is based on a research grant supported by NSF Award IIS-9977336. Several members of the M5 Research Group at the School of Electrical Engineering and Computer Science of University of Central Florida participated in this research. The contributions of Dr. Robert Franceschini and Mr. Holger Kruse with respect to Star transform work are acknowledged. The collaboration of Mr. Nitin Motgi and Mr. Nan Zhang in obtaining some of the experimental results is also gratefully acknowledged. Mr. Nitin Motgi’s help in developing the compression utility web site is also acknowledged.

10.7 REFERENCES

1. Arnavut, Z., 2000. Move-to-front and inversion coding. *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 193–202.
2. Awan, F., and A. Mukherjee, 2001. LIPT: A lossless text transform to improve compression. *International Conference on Information Theory: Coding and Computing*, Las Vegas, NV. IEEE Comput. Soc., Los Alamitos, CA. pp. 452–460, April 2001.

3. Awan, F., 2001. *Lossless Reversible Text Transforms*, M.S. thesis, University of Central Florida, Orlando, July 2001.
4. Balkenhol, B., S. Kurtz, and Y. M. Shtarkov, 1999. Modifications of the Burrows Wheeler data compression algorithm. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 188–197.
5. Bell, T. C., and A. Moffat, 1989. A note on the DMC data compression scheme. *The British Computer Journal*, Vol. 32, No. 1, pp. 16–20.
6. Burrows, M., and D. J. Wheeler, 1994. A Block-Sorting Lossless Data Compression Algorithm. SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA.
7. Bzip2 Memory Usage. Available at <http://krypton.mnnsu.edu/krypton/software/bzip2.html>.
8. Calgary and Canterbury Corp. Available at <http://corpus.canterbury.ac.nz>.
9. Chapin, B., 2000. Switching between two on-line list update algorithms for higher compression of Burrows–Wheeler transformed data. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 183–191.
10. Cleary, J. G., and I. H. Witten, 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions and Communications*, Vol. COM-32, No. 4, pp. 396–402.
11. Cleary, J. G., W. J. Teahan, and I. H. Witten, 1995. Unbounded length contexts for PPM. In *Proceedings of Data Compression Conference*, pp. 52–61, March 1995.
12. Cormack, G. V., and R. N. Horspool, 1987. Data compression using dynamic Markov modeling. *Computer Journal*, Vol. 30, No. 6, pp. 541–550.
13. Effros, M., 2000. PPM Performance with BWT complexity: A new method for lossless data compression. *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 203–212.
14. Fenwick, P., 1996. Block sorting text compression. In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia, January 31–February 2, 1996*.
15. Franceschini, R., and A. Mukherjee, 1996. Data compression using encrypted text. In *Proceedings of the Third Forum on Research and Technology, Advances on Digital Libraries, ADL 96*, pp. 130–138.
16. Gibson, J. D., T. Berger, T. Lookabaugh, D. Lindbergh, and R. L. Baker, 1998. Digital Compression for Multimedia: Principles and Standards. Morgan Kaufmann, San Mateo, CA.
17. Available at <http://www.promo.net/pg/>.
18. Howard, P. G., 1993. *The Design and Analysis of Efficient Lossless Data Compression Systems*, Ph.D. thesis. Brown University, Providence, RI.
19. Huffman, D. A., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, Vol. 40, pp. 1098–1101.
20. Kruse, H., and A. Mukherjee, 1997. Data compression using text encryption. In *Proceedings of Data Compression Conference*, p. 447, IEEE Comput. Soc., Los Alamitos, CA.
21. Nelson, M. R., “Star Encoding”, *Dr. Dobb’s Journal*, August, 2002, pp. 94–96.
22. Larsson, N. J., 1998. The Context Trees of Block Sorting Compression. N. Jesper Larsson: The context trees of block sorting compression. In *Proceedings of Data Compression Conference*, pp. 189–198.
23. Moffat, A., 1990. Implementing the PPM data compression scheme, *IEEE Transactions on Communications*, Vol. 38, No. 11, pp. 1917–1921.
24. PPMDH. Available at <ftp://ftp.elf.stuba.sk/pub/pc/pack/>.
25. RK archiver. Available at <http://rksoft.virtualave.net/>.
26. RK archiver. Available at <http://www.geocities.com/SiliconValley/Lakes/1401/compress.html>.
27. Sadakane, K., T. Okazaki, and H. Imai, 2000. Implementing the context tree weighting method for text compression. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 123–132.
28. Salomon, D., 2000. *Data Compression: The Complete Reference*, 2nd ed. Springer-Verlag, Berlin/New York.
29. Sayood, K., 1996. *Introduction to Data Compression*. Morgan Kaufman, San Mateo, CA.
30. Seward, J., 2000. On the performance of BWT sorting algorithms. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 173–182.
31. Shannon, C. E., and W. Weaver, 1998. *The Mathematical Theory of Communication*. Univ. of Illinois Press, Champaign.
32. Shannon, C. E., 1948. A mathematical theory of communication. *Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656.

33. Available at <http://trec.nist.gov/data.html>.
34. Willems, F., Y. M. Shtarkov, and T. J. Tjalkens, 1995. The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, Vol. IT-41, No. 3, pp. 653–664.
35. Witten, I. H., A. Moffat, and T. Bell, 1999. *Managing Gigabyte, Compressing and Indexing Documents and Images*, 2nd ed. Morgan Kaufmann, San Mateo, CA.
36. YBS Compression Algorithm. Available at <http://arrest1.tripod.com/texts18.html>.
37. Ziv, J., and A. Lempel, 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pp. 337–343, May 1977.

This Page Intentionally Left Blank

Compression of Telemetry

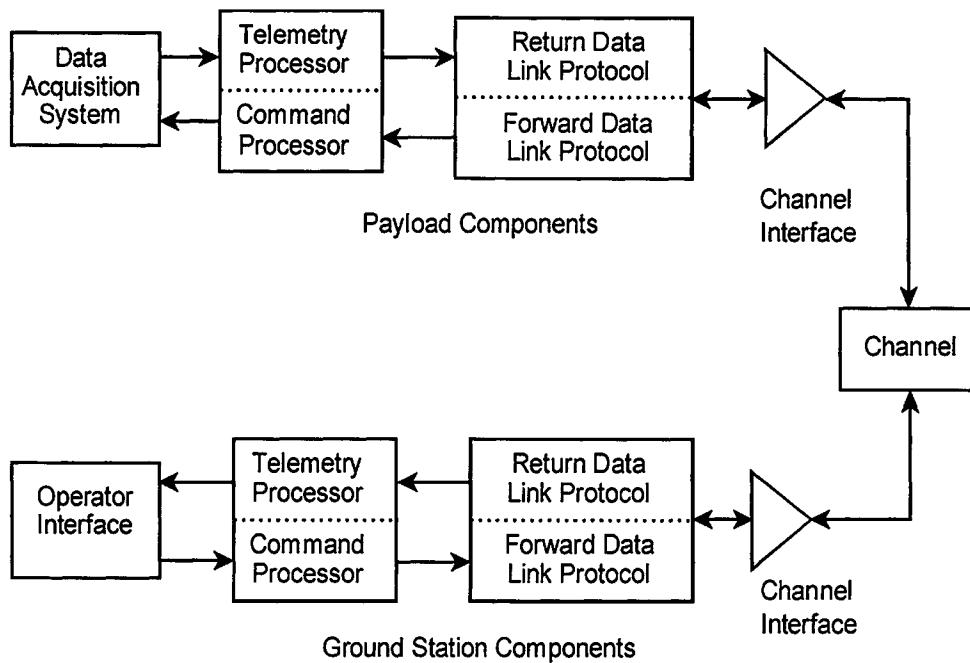
SHEILA HORAN

OVERVIEW

Telemetry compression is not a well-defined problem. Each application of compression needs to address different data types. This wide variety has made a single compression technique impossible. Telemetry now encompasses every type of data. The only way to approach compression for this type of problem will involve hybrid techniques. Telemetry has seemed to resist compression, but its time has come.

11.1 WHAT IS TELEMETRY?

According to the National Telecommunications and Information Administration's Institute for Telecommunication Sciences, telemetry is "the use of telecommunication for automatically indicating or recording measurements at a distance from the measuring instrument"; it is "the transmission of nonvoice signals for the purpose of automatically indicating or recording measurements at a distance from the measuring instrument" [1]. Measurement at a distance then implies that the information is transported by some means from where the data are taken to the user. The form of transportation may be a coaxial cable or a fiber optic cable or the data may be transmitted through space. The distance may be very small or light-years away. However far the information travels, it will use some bandwidth to transmit the information. The content of this information can be almost anything. For the Federal Bureau of Investigations, which conducts covert operations, the information gathered could be sounds or voices. In some cases the information is the reflected signal of radar, sonar, or even X rays. Originally the term referred to the housekeeping functions of spacecraft or aircraft. Measurements of the craft's vital data (and human medical readings) were taken, organized, and then transmitted by packets or frames

**FIGURE 11.1**

Satellite telemetry system [3].

on a special link received and then monitored. In the first method, time division multiplexing, the measurements are either buffered or integrated into the data to be transmitted. In the second method, frequency division multiplexing, the measurements are sent at particular frequencies. An example of a satellite telemetry system can be seen in Fig. 11.1.

Packet format is flexible in the type of data to be sent. The data are sent in a burst and have an address heading indicating where their destination is. This allows the use of different packet types each time the data are sent. Figure 11.2 shows the possible flow of the data into the packet format.

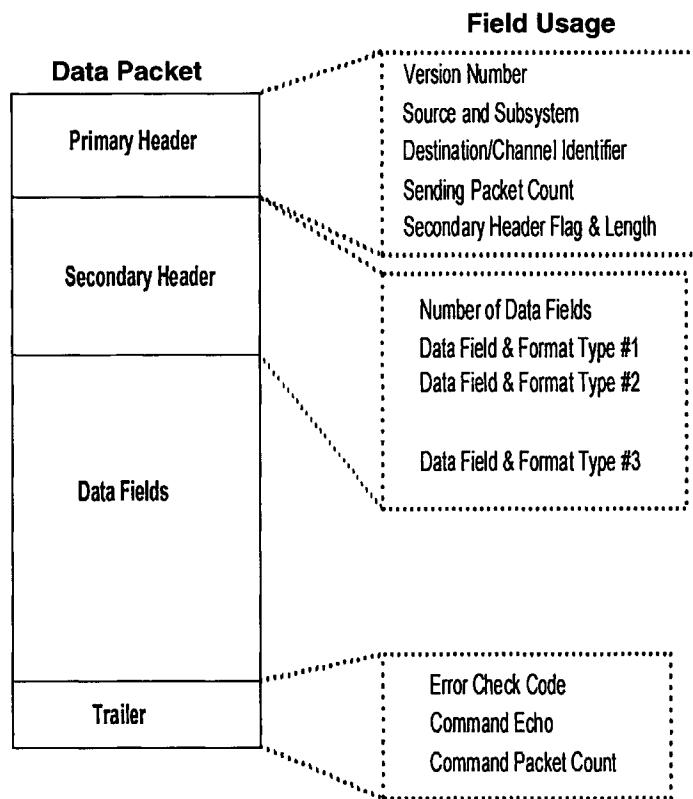
Telemetry frames have more of a definite fixed structure and tend to be used when lots of data will be sent with multiple measurements over time. In each frame, the measurements are usually in the same place in the frame.

In general the payload would be the sensor(s), or the device that accumulates data, and the ground station is where the data would be processed. The channel refers to the transmission medium between where the data are acquired and where the data are delivered. There are two main formats for telemetry data. The data are either put into packets (used especially for Internet users) or put into telemetry frames. Examples of the packet telemetry are shown in Figs. 11.2 and 11.3, and examples of frame telemetry are shown in Figs. 11.4 and 11.5.

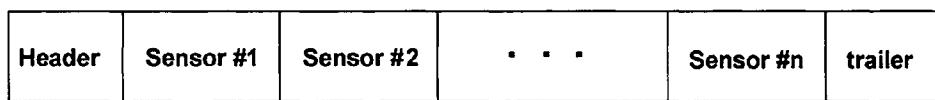
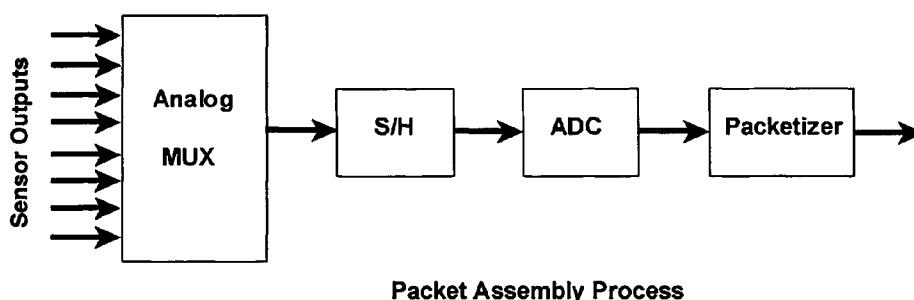
The internal ordering of packet or frame telemetry can change for different uses, but the outer structure of each is the same no matter what the application.

Today, telemetry includes images and voice data as well as the housekeeping type of data.

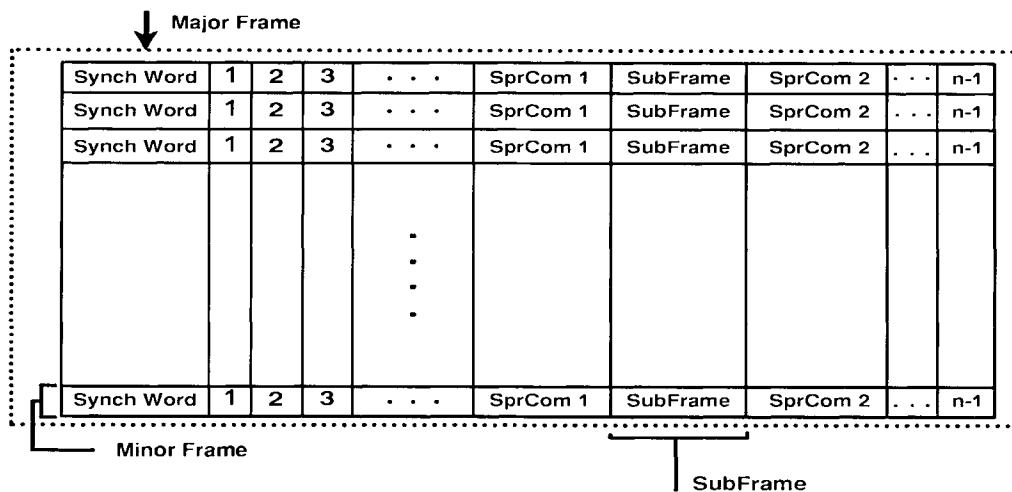
Some of the telemetry used by the Air Force interleaves the pilot's voice data with the aircraft monitoring data. Each military exercise that is performed has its own telemetry order. Hence every telemetry data packet or frame is unique. This adds to the complications of compression. Each test could require a different compression technique.

**FIGURE 11.2**

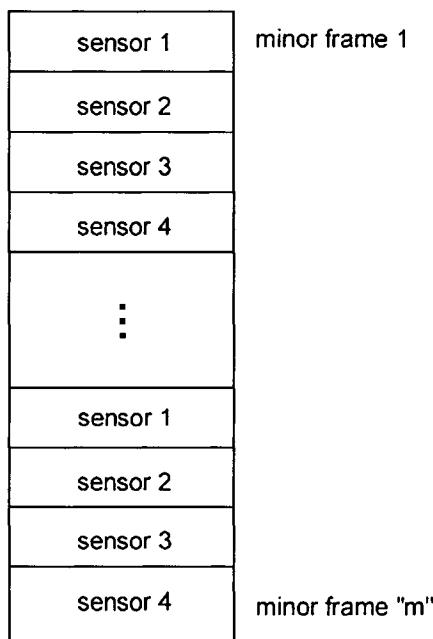
Packet structure for telemetry data [3].

**Packet Format****FIGURE 11.3**

Packet telemetry format [3].

**FIGURE 11.4**

Major frame structure for telemetry data [3].

**FIGURE 11.5**

Minor frame structure for telemetry frames [3].

11.2 ISSUES INVOLVED IN COMPRESSION OF TELEMETRY

11.2.1 Why Use Compression on Telemetry

Bandwidth is a precious commodity. Bandwidth is the amount of frequency space that a signal takes to transmit its information. There are several different definitions of bandwidth. Two common definitions are the 3-dB bandwidth and the null-to-null bandwidth. For a given power spectrum of a signal, the 3-dB bandwidth is where the signal power is down 3 dB from the maximum.

The null-to-null bandwidth is the amount of frequency that is between the first two nulls of the signal (the first low points from the maximum). Given that the bit rate = R = (number of data bits sent)/s, the bandwidths for binary-phase shift keying (a common type of modulation used to transmit data) are [2] as follows: 3-dB bandwidth, $0.88R$; and null-to-null bandwidth, $1.0R$.

If the number of data bits was reduced by half, then one would need only half the bandwidth to transmit the data or one could then send twice as much in the same amount of bandwidth.

With the increase in use of wireless technology, it has become imperative that bandwidth be used efficiently. To this end, data need to be reduced or modulation techniques need to be used to minimize the required bandwidth. Data compression (coding) is one technique to reduce the bandwidth needed to transmit data.

The goals of data coding are as follows:

- Reduce the amount of data;
- Find codes that take into account redundancy, structure, and patterns;
- Break up data into random groups.

Since telemetry data measure quantities that are unknown (as in engine temperature, pressures, speeds, etc.), the need for lossless data reception is important. One of the greatest fears of anyone conducting experiments or receiving data from anywhere is that the data might become corrupted or be lost. When data are compressed, there is less redundancy in the data. This makes it harder to retrieve data once they have been corrupted. When something unexpected happens, e.g., a test goes wrong, these measurements are extremely important to help the engineers understand what happened. So, along with the lossless data compression, channel coding may be needed to ensure no loss of data. What can be done is to channel code the compressed data to protect them against errors. But when channel coding is added, the amount of data is increased. If an appropriate channel/source-coding scheme is selected, then the compressed data with the channel coding combined will still be less data than there would have been originally.

To achieve bandwidth reduction or data reduction (for storage requirements), the compression technique needs to losslessly compress the data taking into account data structure, patterns, and redundancy.

11.2.2 Structure of the Data

The data structure of the frame or packet can be used to advantage. After the initial frame of data is sent, subsequent frames could be sent as a difference in the frames. If the location of the measurements in the frame remains the same from frame to frame, then the values will generally stay within a given range. This means that the differences between the frames would be small, giving smaller numerical values to send. Smaller sized numbers just by themselves can lead to compression. The resulting frame structure would remain the same, but the frame itself would be smaller in size.

11.2.3 Size Requirements

With the advances that have been made in technology, the processor needed to perform the compression could be the size of a single chip. The interfaces to connect the chip to the original process could all be placed on just one card added to the data collection device. The weight would amount to only ounces. The hard part is not in building the device to do the compression, but in deciding what compression to use.

11.3 EXISTING TELEMETRY COMPRESSION

There are many different types of data compression algorithms in place. The first level of data compression comes in deciding how often to take the samples. How often to take the measurements depends on the rapidity with which the data change. The quicker the change (the higher the frequency of change), the faster the data need to be sampled. The slower the change, the slower the data can be sampled. The problem here often is that the scientists and engineers may not know in advance how quickly the signal will change. To account for this, most data are oversampled. If the signal can be monitored, and the sampling changed as needed, this could optimize the number of samples taken. The Nyquist Theorem states that a band-limited signal (with band limit B) can be correctly recovered if the signal is sampled at most every t_S seconds, where $t_S = 1/(2B)$. If in addition there is noise present in the signal (with a signal-to-noise ratio of 10), then a more realistic sampling time would be $t_S = 1/(5B)$ [3].

Much of the current telemetry is images. When images are the information being transmitted, then the compression techniques for images are the appropriate ones to use. When voice data are being transmitted, then accepted techniques for voice compression are used. The challenge comes when many different types of data are interspersed in the data to be sent. In this case, either a frame to frame (or packet to packet) differencing can be done or the format of the data can be reorganized to take advantage of the structure of the data. Table 11.1 lists some of the current applications of compression for telemetry data. It is clear that there are a wide variety of compression techniques. Each type of data is different and so requires a different type of compression scheme.

Table 11.1 Summary of Compression Techniques Used for Telemetry

Device	Compression Used	Reference
SPOT 4 image optical & radar remote sensing satellite	DPCM for imaging	[4]
ECG data compression	First-order prediction	[5]
EUMETSAT meteorological satellite	Lossless wavelet transform JPEG	[6]
COBE	Vector quantization, modified Huffman, runlength code, Chebyshev, Rice	[7]
Cassini	Walsh transform (images), Rice	[7]
Galileo	Dictionary substitution	[7]
Mars Pathfinder	JPEG	[7]
HST	H COMPRESS	[7]
STS	Block adaptive quantization	[7]
VIS	Hybrid mean residual, vector quantization, runlength, Huffman	[8]
NGST	Cosmic ray removal effects, Rice	[9]
Voyager	Rice	[10]
Space applications	Wavelets for high lossy compression, Rice for lossless compression	[11]
ESA Huygens Titan probe	DCT	[12]
ECG	Transform coding	[13]
Seismic data	First-order predictor, DPCM, wavelets	[13]
Missile/rocket	Zero-order predictor	[13]

Abbreviations: COBE, Cosmic Background Explorer; DPCM, Differential Pulse Code Modulation; ECG, Electrocardiogram; NGST, Next Generation Space Telescope; VIS, Visible Imaging System.

11.4 FUTURE OF TELEMETRY COMPRESSION

As the frequency spectrum becomes more congested, there will be an increased use of data compression. Telemetry data will take advantage of adaptive and hybrid types of data compression. Algorithms similar to the Rice algorithm, in which several types of data compression can be implemented, are the best techniques to use on data in general. More adaptive-type methods will be developed. Since each telemetry application is so specific, no one method of compression will be favored. Each telemetry application will need to utilize the best techniques that address its particular mix of data. Data compression will become a necessity.

11.5 REFERENCES

1. Telecom Glossary, 2000. Available at <http://glossary.its.bldrdoc.gov/projects/telecomglossary2000>.
2. Couch, L., 1993. *Digital and Analog Communication Systems*, p. 115, Table 2-4. Macmillan Co., New York.
3. Horan, S., 1993. *Introduction to PCM Telemetering Systems*. CRC Press, Boca Raton, FL.
4. SPOT Satellite Technical Data. Available at www.spot.com/home/system/introsat/seltec/seltec.htm.
5. Banville, I., and Armstrong, S., 1999. Quantification of real-time ECG data compression algorithms, In *Proceedings of the 1st Joint BMES/EMBS Conference, October 1999, Atlanta, GA*, p. 264. IEEE Publication, New York.
6. MSG Ground Segment LRIT/HRIT Mission Specific Implementation, EUMETSAT document MSG/SPE/057, September 21, 1999.
7. Freedman, I., and P. M. Farrelle, 1996. Systems Aspects of COBE Science Data, July 1996. Available at http://iraf.noao.edu/iraf/web/ADASS/adass_proc/adass_95/freedmani/freedmani.html.
8. Frank, L., J. Sigwarth, J. Craven, J. Cravens, J. Dolan, M. Dvorsky, P. Hardebeck, J. Harvey, and D. Muller, 1993. Visible Imaging System (VIS), December 1993. Available at www-pi.physics.uio.edu/vis/vis_description/node10.html and www-pi.physics.uiowa.edu/.
9. Nieto-Santisteban, M., D. Fixson, J. Offenberg, R. Hanisch, and H. S. Stockman, 1998. Data Compression for NGST. Available at <http://monet.astro.uinc.edu/adass98/Proceedings/nieto-santistebanma>.
10. Gray, R., 1997. Fundamentals of Data Compression, International Conference on Information, Communications, and Signal Processing, Singapore, September 1997. IEEE Publication, New York.
11. Consultative Committee for Space Data Systems (CCSDS), 1997. Lossless Data Compression, May 1997.
12. Ruffer, P., F. Rabe, and F. Gliem, 1992. A DCT image data processor for use on the Huygens Titan probe. In *Proceedings of the IGARSS*, pp. 678–680. IEEE Publication, New York.
13. Lynch, T., 1985. *Data Compression Techniques and Applications*. Reinhold, New York.

This Page Intentionally Left Blank

Lossless Compression of Audio Data

ROBERT C. MAHER

OVERVIEW

Lossless data compression of digital audio signals is useful when it is necessary to minimize the storage space or transmission bandwidth of audio data while still maintaining archival quality. Available techniques for lossless audio compression, or lossless audio *packing*, generally employ an adaptive waveform predictor with a variable-rate entropy coding of the residual, such as Huffman or Golomb–Rice coding. The amount of data compression can vary considerably from one audio waveform to another, but ratios of less than 3 are typical. Several freeware, shareware, and proprietary commercial lossless audio packing programs are available.

12.1 INTRODUCTION

The Internet is increasingly being used as a means to deliver audio content to end-users for entertainment, education, and commerce. It is clearly advantageous to minimize the time required to download an audio data file and the storage capacity required to hold it. Moreover, the expectations of end-users with regard to signal quality, number of audio channels, meta-data such as song lyrics, and similar additional features provide incentives to compress the audio data.

12.1.1 Background

In the past decade there have been significant breakthroughs in audio data compression using lossy *perceptual* coding [1]. These techniques lower the bit rate required to represent the signal by establishing perceptual error criteria, meaning that a model of human hearing perception is

used to guide the elimination of excess bits that can be either reconstructed (redundancy in the signal) or ignored (inaudible components in the signal). Such systems have been demonstrated with “perceptually lossless” performance, i.e., trained listeners cannot distinguish the reconstructed signal from the original with better than chance probability, but the reconstructed waveform may be significantly different from the original signal. Perceptually lossless or near-lossless coding is appropriate for many practical applications and is widely deployed in commercial products and international standards. For example, as of this writing millions of people are regularly sharing audio data files compressed with the MPEG 1 Layer 3 (MP3) standard, and most DVD video discs carry their soundtracks encoded with Dolby Digital multichannel lossy audio compression.

There are a variety of applications, however, in which lossy audio compression is inappropriate or unacceptable. For example, audio recordings that are to be stored for archival purposes must be recoverable bit-for-bit without any degradation, and so lossless compression is required. Similarly, consumers of audio content may choose to purchase and download a losslessly compressed file that provides quality identical to the same music purchased on a conventional audio CD. Furthermore, lossy compression techniques are generally not amenable to situations in which the signal must pass through a series of several encode/decode operations, known as *tandem* encode/decode cycles. This can occur in professional studio applications where multiple audio tracks are additively mixed together or passed through audio effects devices such as EQ filters or reverberators and then reencoded. Tandeming can also occur in broadcasting or content distribution when the signal must be changed from one data format to another or sent through several stages of intermediate storage. Audible degradations due to lossy compression will accumulate with each encode/decode sequence, and this may be undesirable.

12.1.2 Expectations

Lossy audio compression such as MP3 is appropriate for situations in which it is necessary to specify the best perceived audio quality at a specific, guaranteed bit rate. Lossless compression, on the other hand, is required to obtain the lowest possible bit rate while maintaining perfect signal reconstruction. It is important to be aware that the bit rate required to represent an audio signal losslessly will vary significantly from one waveform to another depending on the amount of redundancy present in the signal. For example, a trivial file containing all “zero” samples (perfect silence) would compress down to an integer representing the number of samples in the file, while an audio signal consisting of white noise would thwart any attempt at redundancy removal. Thus, we must be prepared to accept results in which the bit rate of the losslessly compressed data is not significantly reduced compared to the original rate.

Because most audio signals of interest have temporal and spectral properties that vary with time, it is expected that the lossless compression technique will need to adapt to the short-term signal characteristics. The time varying signal behavior will imply that the instantaneous bit rate required to represent the compressed signal will vary with time, too. In some applications, such as storing an audio data file on a hard disk, the major concern is the average bit rate since the size of the resulting file is to be minimized. In other applications, most notably in systems requiring real-time transmission of the audio data or data retrieval from a fixed-bandwidth storage device such as DVD, there may also be concern about the peak bit rate of the compressed data.

A plethora of bit resolutions, sample rates, and multiple channel formats are in use or have been proposed for recording and distribution of audio data. This means that any technique proposed for lossless compression should be designed to handle pulse code modulation (PCM) audio samples with 8- to 32-bit resolution, sample rates up to 192 kHz, and perhaps six or more audio channels. In fact, many of the available commercial lossless compression methods include special features to optimize their performance to the particular format details of the audio data [5].

12.1.3 Terminology

A variety of terms and informal colloquial phrases are used in the description of lossless audio data compression. In this chapter several of these terms may be used interchangeably and it is helpful to keep this in mind so that no special distinction is to be inferred unless specific mention is given in the text. A summary of these terms is given next.

Lossless compression and lossless packing both refer to methods for reducing the number of data bits required to represent a stream of audio samples. Some authors choose to use the term *packing* instead of *compression* in order to avoid potential confusion between the lossy and lossless data compression, and between data compression and dynamic range compression, as in the audio dynamics processing device known as a gain compressor/limiter. In this chapter the full expression “lossless data compression” is used.

The performance of a lossless data compressor can be interpreted in several ways. One is the *compression ratio*, which is the size of the input data file divided by the size of the output data file. Sometimes the performance is stated as a *percentage*, either the percentage reduction in size or the percentage of the original file size that remains after compression. Another interpretation is to describe data compression as a way to minimize the *bit rate* of the signal, where the bit rate is the number of bits required to represent the data divided by the total playing time. And in some contexts it is useful to describe data compression in terms of the average *number of bits per sample* or the average *reduction in bits per sample*. Clearly, the compression ratio is most helpful when trying to determine how much file system space would be saved by compressing the audio data, while the interpretation in terms of bit rate is more meaningful when considering transmission of the data through a communications channel.

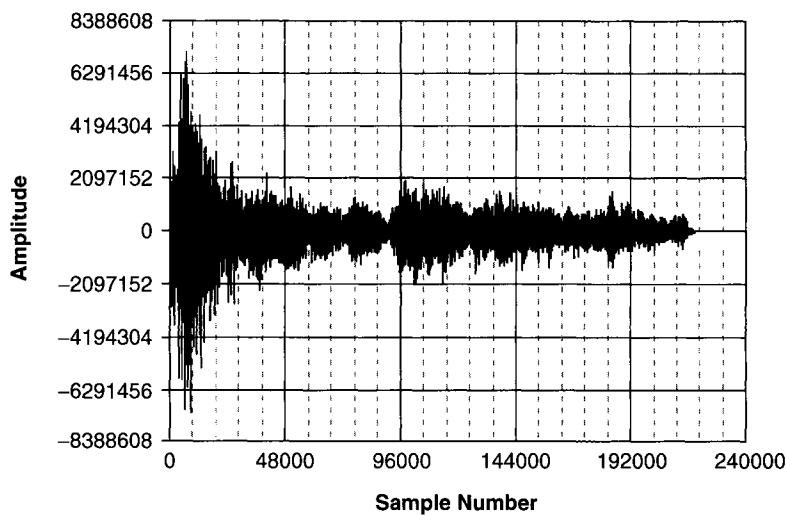
Finally, the act of compressing and decompressing the audio data is sometimes referred to as *encoding* and *decoding*. In this context the encoded data is the losslessly compressed data stream, while the decoded data is the recovered original audio waveform. In this chapter we use *compression* and *encoding* interchangeably.

12.2 PRINCIPLES OF LOSSLESS DATA COMPRESSION

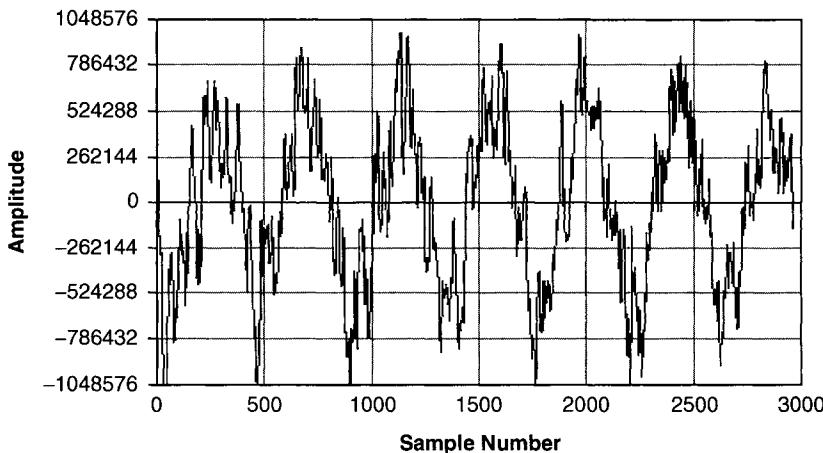
The essence of lossless data compression is to obtain efficient redundancy removal from a bitstream [9]. Common lossless compressors such as WinZIP and StuffIT are used on arbitrary computer data files and usually provide compressed files roughly half the size of the original. However, the compression model (e.g., LZ77) commonly used is poorly matched to the statistical characteristics of binary audio data files, and the compressed audio files are typically still about 90% of the original file size. On the other hand, good audio-specific lossless compressors can achieve output files that are 30–50% of the original file size [4]. Nonetheless, it is important to remember that there are no guarantees about the amount of compression obtainable from an arbitrary audio data file: It is even possible that the compressed file ends up larger than the original due to the overhead of data packing information! Clearly the lossless algorithm should detect this situation and not “compress” that file.

12.2.1 Basic Redundancy Removal

How can an audio-specific lossless compression algorithm work better than a general-purpose Ziv–Lempel algorithm? To examine this question, consider the audio signal excerpt displayed in Fig. 12.1. This excerpt is approximately 5 s of audio taken from a 24-bit PCM data file with 48-kHz sample rate (1.152 million bits/s/channel).

**FIGURE 12.1**

Excerpt (5 s) from an audio recording (48-kHz sample rate, 24-bit samples). Note that on the vertical amplitude scale, $8,388,608 = 2^{23}$; $4,194,304 = 2^{22}$; and $2,097,152 = 2^{21}$.

**FIGURE 12.2**

Enlarged excerpt, approximately 60 ms, from near the middle of the audio recording of Fig. 12.1 (48-kHz sample rate, 24-bit samples). Note that amplitude $1,048,576 = 2^{20}$.

An enlarged portion of the signal in Fig. 12.1 is shown in Fig. 12.2.

The Ziv–Lempel methods take advantage of patterns that occur repeatedly in the data stream. Note that even if the audio waveform is roughly periodic and consistent as in Fig. 12.2, the audio samples generally do not repeat exactly due to the asynchronous relationship between the waveform period and the sample rate, as well as the presence of noise or other natural fluctuations. This property of audio signals makes them quite different statistically from structured data such as English text or computer program source code, and therefore we should not be surprised that different compression strategies are required.

12.2.2 Amplitude Range and Segmentation

Let us do a heuristic look at the excerpt in Fig. 12.1. One characteristic of this signal is that its amplitude envelope (local peak value over a block of samples) varies with time. In fact, a substantial portion of the excerpt has its amplitude envelope below one-quarter of the full-scale value. In this case we know that at least the 2 most significant bits (MSBs) of the PCM word will not vary from sample to sample, other than for changes in sign (assuming 2's complement numerical representation). If we designed an algorithm to identify the portions of the signal with low amplitude, we could obtain a reduction in bit rate by placing a special symbol in the data stream to indicate that the n most significant bits are the same for each of the following samples, and thus we can eliminate the redundant n MSBs from each subsequent sample [2]. We could also go further by indicating somehow that several samples in a row would share the same sign. Of course, we would need to monitor the signal to detect when the low-amplitude condition was violated, so in addition to the “same MSB” symbol we might also choose to include an integer indicating the number of samples, or block size, for which the condition remained valid. Thus, we could expect to get a reduction in bit rate by *segmentation* of the signal into blocks where the signal’s amplitude envelope was substantially less than full scale. It is clear, of course, that the amount of compression will depend upon how frequently low-amplitude blocks occur in the signal.

We can also examine another amplitude-related feature of audio data. As mentioned above, the audio excerpt shown in Figs. 12.1 and 12.2 was obtained from a data file of 24-bit PCM samples. However, if we look carefully at the data for this example, we can discover another heuristic strategy for data compression [2]. Some of the audio data sample values taken from near the beginning of the segment shown in Fig. 12.2 are given here as numerical values:

Sample No.	Decimal Value	Binary Value (24-bit 2's complement)
0	91648	00000001:01100110:00000000
1	95232	00000001:01110100:00000000
2	107008	00000001:10100010:00000000
3	89344	00000001:01011101:00000000
4	42240	00000000:10100101:00000000
5	-19456	11111111:10110100:00000000
6	-92672	11111110:10010110:00000000
7	-150784	11111101:10110011:00000000
8	-174848	11111101:01010101:00000000
9	-192512	11111101:00010000:00000000
10	-215552	11111100:10110110:00000000
11	-236544	11111100:01100100:00000000
12	-261120	11111100:00000100:00000000
13	-256000	11111100:00011000:00000000
14	-221440	11111100:10011111:00000000
15	-222208	11111100:10011100:00000000

As discussed above, note that the MSBs of this excerpt do not carry much information. In fact, this group of 16 samples is low enough in amplitude that the 6 most significant bits can be replaced by a single bit indicating the sign of the number, giving a tidy bit rate reduction of 5 bits per sample. Looking further, notice that even though the data file is stored with 24-bit resolution, the actual data samples do not contain information in the 8 least significant bits (LSBs). This is because the original data stream was actually obtained from a digital audio tape that included only the standard 16-bit per sample audio resolution, and thus the 8 LSBs are filled with zeros.

Again, we can obtain significant data compression merely by storing a special format code that identified the m LSBs (8 in this case) as being redundant for the file and not actually store them. The decoder would detect the special format code and reinsert the m LSBs in the decoded output file. Although such a distinct situation of excess data resolution might seem contrived—and we certainly cannot count on this peculiarity in all data files—it is mentioned here as an example of some of the special circumstances that can be detected to aid in lossless audio compression.

12.2.3 Multiple-Channel Redundancy

Common audio data formats such as the Compact Disc generally store a two-channel (stereo) audio signal. The left and right channel data streams are stored entirely independently. Various new data formats for DVD and other distribution systems will allow four, or six, or perhaps more separate audio channels for use in multichannel surround loudspeaker playback systems. In any case, it is common to find that there is at least some correlation between the two channels in a stereo audio signal or among the various channels in a multichannel recording, and therefore it may be possible to obtain better compression by operating on two or more channels together [3, 6]. This is referred to as *joint stereo coding* or *interchannel coding*. Joint coding is particularly useful for audio signals in which two or more of the channels contain the same signal, such as a dual-mono program, or when one or more of the channels is completely silent for all or most of the recording. Some systems take advantage of interchannel correlation by using a so-called *matrixing* operation to encode L and $(L - R)$, or perhaps $(L + R)$ and $(L - R)$, which is similar to FM broadcast stereo multiplexing. If the L and R channels are identical or nearly identical, the difference signal $(L - R)$ will be small and relatively easy to encode.

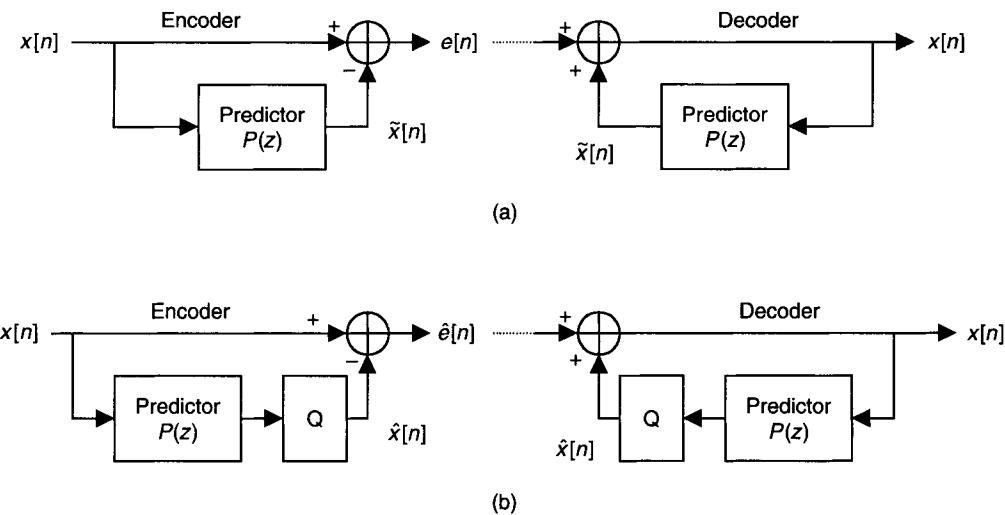
12.2.4 Prediction

In many cases of practical interest audio signals exhibit a useful degree of sample-to-sample correlation. That is, we may be able to *predict* the value of the next audio sample based on knowledge of one or more preceding samples [7–9]. Another way of stating this is that if we can develop a probability density function (PDF) for the *next* sample based on our observation of previous samples, and if this PDF is non-uniform and concentrated around a mean value, we will benefit by storing only (a) the minimum information required for the decoder to re-create the same signal estimate and (b) the error signal, or *residual*, giving the sample-by-sample discrepancy between the predicted signal and the actual signal value. If the prediction is very close to the actual value, the number of bits required to encode the residual will be fewer than the original PCM representation. In practice the signal estimate is obtained using some sort of adaptive linear prediction.

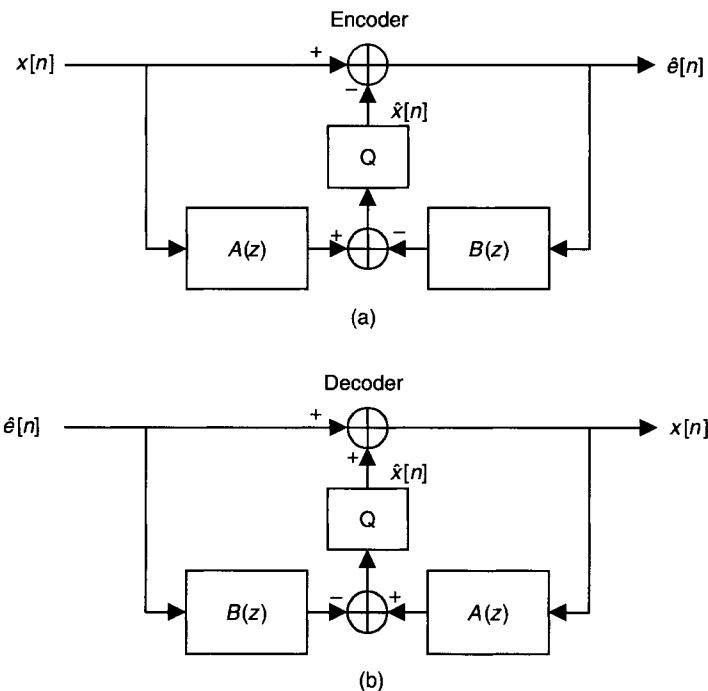
A general model of a signal predictor is shown in Fig. 12.3a [3, 7]. The input sequence of audio samples, $x[n]$, serves as the input to the prediction filter, $P(z)$, creating the signal estimate $\tilde{x}[n]$. The prediction is then subtracted from the input, yielding the error residual signal, $e[n]$.

In a practical prediction system it is necessary to consider the numerical precision of the filter $P(z)$, since the coefficient multiplications within the filter can result in more significant bits in $\tilde{x}[n]$ than in the input signal $x[n]$. This is undesirable, of course, since our interest is in minimizing the output bit rate, not adding more bits of precision. The typical solution is to quantize the predicted value to the same bit width as the input signal, usually via simple truncation [2]. This is indicated in Fig. 12.3b. As will be mentioned later in the section on practical design issues, special care must be taken to ensure that the compression and decompression calculations occur identically on whatever hardware platform is used.

An alternative predictor structure that is popular for use in lossless audio compression is shown in Fig. 12.4 [3]. This structure incorporates two finite impulse response (FIR) filters, $A(z)$ and

**FIGURE 12.3**

(a) Basic predictor structure for lossless encoder/decoder. (b) Structure with explicit quantization (truncation) to original input bit width.

**FIGURE 12.4**

Alternative predictor structure.

$B(z)$, in a feed-forward and feed-back arrangement similar to an infinite impulse response (IIR) Direct Form I digital filter structure, but with an explicit quantization prior to the summing node. Note also that filter $B(z)$ can be designated as a null filter, leaving the straightforward FIR predictor of Fig. 12.3b. The advantage of selecting $A(z)$ and $B(z)$ to be FIR filters is that the coefficients can be quantized easily to integers with short word lengths, thereby making an integer implementation possible on essentially any hardware. Because this structure is intended for signal prediction and not to approximate a specific IIR filter transfer function, the details of the coefficients in $A(z)$ and $B(z)$ can be defined largely for numerical convenience rather than transfer function precision.

The use of an FIR linear predictor ($B(z) = 0$) is quite common for speech and audio coding, and the filter coefficients for $A(z)$ are determined in order to minimize the mean-square value of the residual $e[n]$ using a standard linear predictive coding (LPC) algorithm [7, 8]. No such convenient coefficient determination algorithm is available for the IIR predictor ($B(z) \neq 0$), which limits the widespread use of the adaptive IIR version. In lossless compression algorithms that utilize IIR predictors it is common to have multiple sets of fixed coefficients from which the encoder chooses the set that provides the best results (minimum mean-square error) on the current block of audio samples [3, 5]. In fact, several popular lossless compression algorithms using FIR prediction filters also include sets of fixed FIR coefficients in order to avoid the computational cost of calculating the optimum LPC results [7].

Once the encoder has determined the prediction filter coefficients to use on the current block, this information must be conveyed to the decoder so that the signal can be recovered losslessly. If an LPC algorithm is used in the encoder, the coefficients themselves must be sent to the decoder. On the other hand, if the encoder chooses the coefficients from among a fixed set of filters, the encoder needs only to send an index value indicating which coefficient set was used.

The choice of predictor type (e.g., FIR vs IIR), predictor order, and adaptation strategy has been studied rather extensively in the literature [7]. Several of the lossless compression packages use a low-order linear predictor (order 3 or 4), while some others use predictors up to order 10. It is interesting to discover that there generally appears to be little additional benefit to the high-order predictors, and in some cases the low-order predictor actually performs better. This may seem counterintuitive, *but keep in mind that there often is no reason to expect that an arbitrary audio signal should fit a predictable pattern*, especially if the signal is a complex combination of sources such as a recording of a musical ensemble.

12.2.5 Entropy Coding

After the basic redundancy removal steps outlined above, most common lossless audio compression systems pass the intermediate predictor parameters and prediction error residual signal through an entropy coder such as Huffman, Golomb–Rice, or run length coding [8, 9].

The appropriate statistical model for the prediction error residual signal is generally well represented by a Laplacian probability density function. The Laplacian PDF $l(x)$ is given by

$$l(x) = \frac{1}{\sqrt{2}\sigma} e^{-\frac{\sqrt{2}}{\sigma}|x|},$$

where σ^2 is the variance of the source distribution and $|x|$ is the absolute value (magnitude) of x (in our case x is the error residual signal, $e[n]$).

In order for the entropy coder to be matched to the Laplacian distributed source the coding parameters must be matched to the assumed or estimated variance of the error residual signal. For example, in an optimal Huffman code we would like to select the number of least significant bits, m , so that the probability of generating a code word $m + 1$ bits long is 0.5, the probability

of generating a code word $m + 2$ bits long is $2^{-(m+1)}$, and so forth. This gives [8]:

$$\begin{aligned} m &= \log_2 \left(\log(2) \frac{\sigma}{\sqrt{2}} \right) \\ &= \log_2 (\log(2) E(|x|)), \end{aligned}$$

where $E(\cdot)$ is the expectation operator. Thus, if we determine empirically the expected value of the error residual signal for a segment of data, we can choose a reasonable value of m for the entropy code on that segment. Details on Huffman coding are provided in Chapter 3. Arithmetic coding is discussed in Chapter 4.

12.2.6 Practical System Design Issues

Given the basic rationale for lossless audio coding, we should now consider some of the implementation details that will provide a practical and useful system.

12.2.7 Numerical Implementation and Portability

It is highly desirable in most applications to have the lossless compression and decompression algorithms implemented with identical numerical behavior on any hardware platform. This might appear trivial to achieve if the algorithm exists in the form of computer software, but ensuring that the same quantization, rounding, overflow, underflow, and exception behavior occurs on any digital processor may require special care. This is particularly true if the word size of the audio data exceeds the native word size of the processor, such as 20-bit audio samples being handled on a DSP chip with 16-bit data registers. Similarly, care must be taken if the compression/decompression algorithm is implemented using floating point arithmetic since floating point representations and exception handling may vary from one processor architecture to another.

Most commercially available lossless compression software packages appear to address the arithmetic issue by computing intermediate results (e.g., prediction filter output) with high precision, then quantize via truncation to the original integer word size prior to the entropy coding step [7]. As long as the target processors handle arithmetic truncation properly, the results will be compatible from one processor to another.

12.2.8 Segmentation and Resynchronization

In a practical lossless compression system we will need to break up the input audio stream into short segments over which the signal characteristics are likely to be roughly constant. This implies that a short block size is desired. However, the prediction filter coefficients will probably be determined and sent to the decoder for each block, indicating that we would want the block size to be relatively long to minimize the overhead of transmitting the coefficients. Thus, we will need to handle a trade-off between optimally matching the predictor to the varying data (short block length is better) and minimizing the coefficient transmission overhead (long block is better). In commercially available systems the block length is generally between 10 and 30 ms (around 1000 samples at a 44.1-kHz sample rate), and this appears to be a reasonable compromise for general audio compression purposes [7].

In many applications it may be required—or at least desirable—to start decoding the compressed audio data at some point other than the very beginning of the file without the need to decode the entire file up to that point. In other words, we may want to jump ahead to a particular edit point to extract a sound clip, or we may want to synchronize or time-align several different

recordings for mixdown purposes. We may also need to allow rapid resynchronization of the data stream in the event of an unrecoverable error such as damaged or missing data packets in a network transmission. Since the bit rate varies from time to time in the file and the entropy coding will also likely be of variable length, it is difficult to calculate exactly where in the file to jump in order to decode the proper time segment. Thus, in a practical lossless compression system it is necessary for the encoder to provide framing information that the decoder can use to determine where in the uncompressed time reference a particular block of compressed data is to be used. The compressed data stream must be designed in such a way to meet the practical framing requirements while still obtaining a good compression ratio [5].

12.2.9 Variable Bit Rate: Peak versus Average Rate

As mentioned previously, the characteristics of typical audio signals vary from time to time and therefore we must expect the required bit rate for lossless compression to vary as well. Since the bit rate will vary, a practical lossless system will be described by both an *average* and a *peak* bit rate. The average rate is determined by dividing the total number of bits in the losslessly compressed stream by the total audio playing time of the uncompressed data. The peak bit rate is the maximum number of bits required for any short-term block of compressed audio data, divided by the playing time of the uncompressed block of audio. It might appear that the peak bit rate would never exceed the original bit rate of the digital audio signal, but because it is possible that the encoder must produce prediction filter coefficients, framing information, and other miscellaneous bits along with the data stream itself, the total overhead may actually result in a higher peak bit rate.

The average compressed bit rate spec is most relevant for storage of audio files on computer hard disks, data tapes, and so forth, since this specifies the reduction in the storage space allocated to the file. We obviously would like the average bit rate to be lower than the rate of the original signal, otherwise we will not obtain any overall data compression. Most published comparisons of lossless audio compression systems compare the average compressed bit rate, because this is often the most relevant issue for an audio file that is simply going to be packed into a smaller space for archival purposes [4, 7].

Nonetheless, the peak bit rate is a significant issue for applications in which the short-term throughput of the system is limited [2]. This situation can occur if the compressed bit stream is conveyed in real time over a communication channel with a hard capacity limitation, or if the available buffer memory in the transmitter and/or receiver does not allow the accumulation of enough excess bits to accommodate the peak bit rate of the compressed signal. The peak bit rate is also important when the compressed data are sent over a channel shared with other accompanying information such as compressed video, and the two compressed streams need to remain time-synchronized at all times. Thus, most commercial lossless audio data compression systems employ strategies to reduce the peak-to-average ratio of the compressed bit stream [5].

12.2.10 Speed and Complexity

In addition to the desire for the greatest amount of compression possible, we must consider the complexity of the compression/decompression algorithms and the feasibility of implementing these algorithms on the available hardware platforms. If we are interested mainly in archiving digital audio files that are stored on hard disks or digital tape, it is natural simply to use a compression/decompression computer program. Ideally we would like the computer program to be sufficiently fast that the compression and decompression procedures would be limited by the access rate of the hard disk drive, not the computation itself.

In some applications the audio content producer may wish to distribute the compressed recordings to mass-market consumers, perhaps via the Internet. In this situation the end-user needs only the decompressor: The compression procedure for the recording is performed once “back at the factory” by the content producer. Since only the decompressor is distributed to the consumer, the producer may have the opportunity to choose an *asymmetrical* compression/decompression algorithm in which the complexity and computational cost of the compression process can be arbitrarily high, while the decompression process is made as fast and simple as possible. As long as the decompression algorithm is flexible enough to take advantage of highly optimized bit streams, the content producer has the luxury of iteratively adjusting the prediction coefficients, block sizes, etc., to get the lowest compressed bit rate [5].

12.3 EXAMPLES OF LOSSLESS AUDIO DATA COMPRESSION SOFTWARE SYSTEMS

At the time of this writing, at least a dozen lossless audio compression software packages are available as freeware, shareware, or commercial products [4]. Three examples are summarized next. These packages were chosen to be representative of the current state-of-the-art, but no endorsement should be inferred by their inclusion in this section.

12.3.1 Shorten

The Shorten audio compression software package is based on the work of Tony Robinson at Cambridge University, Cambridge, United Kingdom [8]. The Shorten package provides a variety of user-selectable choices for block length and predictor characteristics. Shorten offers a standard LPC algorithm for the predictor or a faster (but less optimal) mode in which the encoder chooses from among four fixed polynomial predictors. The residual signal after the predictor is entropy encoded using the Rice coding technique. Thus, the Shorten algorithm follows the general lossless compression procedure described above: Samples are grouped into short blocks, a linear predictor is chosen, and the residual signal is entropy coded.

The Shorten software can also be used in a lossy mode by specifying the allowable signal-to-error power level. In this mode the quantization step size for the residual signal is increased. Note that the lossy mode of Shorten is not based on an explicit perceptual model, but simply a waveform error model.

Shorten is distributed as an executable for Microsoft Windows PCs. A free evaluation version and a full-featured commercial version (about \$30) are available for downloading from www.softsound.com. The full version provides a real-time decoder function, the ability to create self-extracting compressed files, and several advanced mode options.

The Shorten application runs as a regular Windows application with a simple graphical user interface. The software allows the user to browse for ordinary Windows audio files (*.wav and raw binary) and then compress/decompress them. The compressed files are assigned the file extension *.shn. Audio files compressed losslessly by Shorten are typically between 40 and 60% of the original file size. The software runs more than 10 times faster than real time on typical PC hardware.

Several other software packages based on the original work of Tony Robinson are also available. These include programs and libraries for Unix/Linux and Macintosh, as well as plug-ins for popular Windows audio players such as WinAmp.

12.3.2 Meridian Lossless Packing (MLP)

The DVD Forum, an industry group representing manufacturers of DVD recording and playback equipment, has developed a special standard for distribution of high-quality audio material using DVD-style discs. This standard, known as DVD-Audio, provides for up to six separate audio channels, sample rates up to 192 kHz, and PCM sample resolution up to 24 bits. Typical DVD-Audio discs (single sided) can store nearly 90 min of high-resolution multichannel audio.

Among the special features of the DVD-Audio specification is the option for the content producer to use lossless compression on the audio data in order to extend the playing time of the disc. The specified lossless compression algorithm is called *Meridian Lossless Packing*, invented by Meridian Audio of Cambridge, United Kingdom [5]. MLP was developed specifically for multichannel, multiresolution audio data, and it includes support for a wide range of professional formats, downmix options, and data rate/playing time trade-offs. Consumer electronics devices for playing DVD-Audio discs incorporate the MLP decoder for real-time decompression of the audio data retrieved from the disc.

The MLP system uses three techniques to reduce redundancy. Lossless interchannel decorrelation and matrixing are used to eliminate correlation among the input audio channels. Next, an IIR waveform predictor is selected from a predetermined set of filters in order to minimize intersample correlation for each block. Finally, Huffman coding is used to minimize the bit rate of the residual signals.

Because MLP is designed for use in off-line DVD-Audio production, the compression system operator can use trial and error to find the best combination of sample resolution, bit rate, and channel format to obtain the best signal quality for a given duration of the recording. The complexity of the decoder remains the same no matter how much time the production operator spends selecting the optimum set of compression parameters.

MLP is intended for use in professional audio production and distribution so its designers have incorporated many features for flexibility and reliability. For example, MLP includes full “restart” resynchronization information approximately every 5 ms in the compressed audio stream. This allows the stream to be cued to a particular point in time without decoding the entire prior compressed stream and also allows rapid recovery from serious transmission or storage errors. Another useful professional feature is the ability to include copyright, ownership, and error detection/correction information. MLP also provides a way for the content producer to specify which audio channel is intended for which loudspeaker (e.g., front center, left rear) to achieve the intended audio playback environment.

12.3.3 Sonic Foundry Perfect Clarity Audio (PCA)

The *Perfect Clarity Audio* lossless audio codec uses a combination of an adaptive predictor, stereo interchannel prediction, and Huffman coding of the residual signal [10]. PCA is distributed as part of the proprietary software products from Sonic Foundry, Inc., including Sound Forge, Vegas, and ACID. PCA is designed both for long-term archival backup of audio material and for economical temporary storage of audio tracks that are in the process of being edited or mixed. Because the data are stored losslessly, the user can edit and reedit the material as much as desired without accumulating coding noise or distortion.

The PCA package is intended for mono or stereo audio files with 16-bit or 24-bit PCM sample resolution. Like Shorten and MLP, a compressed file is typically about half the size of the original audio file. The encoding process is sufficiently fast to compress at 10 times real time on typical circa 2001 PC hardware, e.g., 1 min of audio is compressed in about 5 s. The decoding

process requires a lower computation rate than encoding and can be accomplished typically with only a few percent of the available CPU cycles during playback.

PCA incorporates an error detection mechanism to flag any storage or transmission errors that might occur during file handling. PCA also provides the ability to include summary information and accompanying text along with the compressed audio data.

12.4 CONCLUSION

Lossless audio compression is used to obtain the lowest possible bit rate while still retaining perfect signal reconstruction at the decoder. The combination of an adaptive signal predictor with a subsequent lossless entropy coder is the preferred method for lossless audio compression. This is the basic framework utilized in the most popular audio compression software packages. All of the popular lossless audio compression algorithms obtain similar bit rate reductions on real-world audio signals, indicating that the practical limit for lossless audio compression performance has been reached. Typical compressed file sizes are between 40 and 80% of the original file size, which compares rather poorly to the performance of lossy perceptual audio coders which can achieve “perceptually lossless” performance at 10% or less of the original file size. Nonetheless, in applications requiring perfectly lossless waveform coding, the advantages of an audio-specific compressor compared to a general-purpose data compressor are often significant: Audio files compressed with WinZip are typically 80–100% of the original file size.

12.5 REFERENCES

1. Brandenburg, K., 1998. Perceptual coding of high quality digital audio. In *Applications of Digital Signal Processing to Audio and Acoustics* (M. Kahrs and K. Brandenburg, Eds.), Chap. 2, pp. 39–83, Kluwer Academic, Dordrecht/Norwell, MA.
2. Craven, P. G., and Gerzon, M. A., 1996. Lossless coding for audio discs. *Journal of the Audio Engineering Society*, Vol. 44, No. 9, pp. 706–720, September 1996.
3. Craven, P. G., Law, M. J., and Stuart, J. R., 1997. Lossless compression using IIR prediction filters. In *Proceedings of the 102nd Audio Engineering Society Convention, Munich, Germany*, March 1997, Preprint 4415.
4. Dipert, B., 2001. Digital audio gets an audition Part One: Lossless compression. *EDN Magazine*, Jan. 4, 2001, pp. 48–61. Available at <http://www.e-insite.net/ednmag/contents/images/60895.pdf>.
5. Gerzon, M. A., Craven, P. G., Stuart, J. R., Law, M. J., and Wilson, R. J., 1999. The MLP lossless compression system. In *Proceedings of the AES 17th International Conference, Florence, Italy*, September 1999, pp. 61–75.
6. Hans, M., 1998. *Optimization of Digital Audio for Internet Transmission*. Ph.D. Thesis, Georgia Institute of Technology, May 1998, Available at <http://users.ece.gatech.edu/~hans/thesis.zip>.
7. Hans, M., and Schafer, R. W., 1999. Lossless Compression of Digital Audio, Hewlett-Packard Technical Report HPL-1999-144, November 1999, Available at <http://www.hpl.hp.com/techreports/1999/HPL-1999-144.html>.
8. Robinson, T., 1994. SHORTEN: Simple Lossless and Near-Lossless Waveform Compression, Technical Report CUED/F-INFENG/TR.156, Cambridge University Engineering Department, Cambridge, UK, December 1994, Available at http://svr-www.eng.cam.ac.uk/reports/svr-ftp/robinson_tr156.ps.Z.
9. Sayood, K., 2000. *Introduction to Data Compression*, 2nd ed., Morgan Kaufmann.
10. Sonic Foundry, Inc., 2001. Private communication, Madison, WI, May 2001.

This Page Intentionally Left Blank

Algorithms for Delta Compression and Remote File Synchronization

TORSTEN SUEL
NASIR MEMON

OVERVIEW

Delta compression and remote file synchronization techniques are concerned with efficient file transfer over a slow communication link in the case where the receiving party already has a similar file (or files). This problem arises naturally, e.g., when distributing updated versions of software over a network or synchronizing personal files between different accounts and devices. More generally, the problem is becoming increasingly common in many network-based applications where files and content are widely replicated, frequently modified, and cut and reassembled in different contexts and packagings.

In this chapter, we survey techniques, software tools, and applications for delta compression, remote file synchronization, and closely related problems. We first focus on delta compression, where the sender knows all the similar files that are held by the receiver. In the second part, we survey work on the related, but in many ways quite different, problem of remote file synchronization, where the sender does not have a copy of the files held by the receiver.

13.1 INTRODUCTION

Compression techniques are widely used in computer networks and data storage systems to increase the efficiency of data transfers and reduce space requirements on the receiving device. Most techniques focus on the problem of compressing individual files or data streams of a certain type (text, images, audio). However, in today's network-based environment it is often the case that files and content are widely replicated, frequently modified, and cut and reassembled in different contexts and packagings.

Thus, there are many scenarios where the receiver in a data transfer already has an earlier version of the transmitted file or some other file that is similar, or where several similar files are transmitted together. Examples are the distribution of software packages when the receiver already has an earlier version, the transmission of a set of related documents that share structure or content (e.g., pages from the same web site), or the remote synchronization of a database. In these cases, we should be able to achieve better compression than that obtained by individually compressing each file. This is the goal of the delta compression and remote file synchronization techniques described in this chapter.

Consider the case of a server distributing a software package. If the client already has an older version of the software, then an efficient distribution scheme would send only a patch to the client that describes the differences between the old and the new versions. In particular, the client would send a request to the server that specifies the version number of the outdated version at the client. The server then looks at the new version of the software, and at the outdated version that we assume is available to the server, and computes and sends out a “patch” that the client can use to update its version. The process of computing such a patch of minimal size between two files is called *delta compression*, or sometimes also *delta encoding* or *differential compression*.

Of course, in the case of software updates these patches are usually computed off-line using well-known tools such as *bdiff*, and the client can then choose the right patch from a list of files. However, *bdiff* is not a very good delta compressor, and there are other techniques that can result in significantly smaller patch size.

When distributing popular software that is updated only periodically, it seems realistic to assume that the server has copies of the previous versions of the software which it can use to compute a delta of minimal size. However, in other scenarios, the server may have only the new version, due to the overhead of maintaining all outdated versions or due to client-side or third-party changes to the file. The *remote file synchronization* problem is the problem of designing a protocol between the two parties for this case that allows the client to update its version to the current one while minimizing communication between the two parties.

13.1.1 Problem Definition

More formally, we have two strings (files) $f_{new}, f_{old} \in \Sigma^*$ over some alphabet Σ (most methods are character/byte oriented) and two computers, C (the client) and S (the server), connected by a communication link.

- In the *delta compression* problem, C has a copy of f_{old} and S has copies of both f_{new} and f_{old} , and the goal for S is to compute a file f_δ of minimum size, such that C can reconstruct f_{new} from f_{old} and f_δ . We also refer to f_δ as a *delta* of f_{new} and f_{old} .
- In the *remote file synchronization* problem, C has a copy of f_{old} and S has only a copy of f_{new} , and the goal is to design a protocol between the two parties that results in C holding a copy of f_{new} , while minimizing the communication cost.

We also refer to f_{old} as a *reference file* and to f_{new} as the *current file*. For a file f , we use $f[i]$ to denote the i th symbol of f , $0 \leq i < |f|$, and $f[i, j]$ to denote the block of symbols from i until (and including) j . We note that while we introduce the delta compression problem here in a networking context, another important application area is in the space-efficient storage of similar files, e.g., multiple versions of a document or a software source—in fact, delta compression techniques were first introduced in the context of software revision control systems. We discuss such applications in Subsection 13.2.1, and it should be obvious how to adapt the definitions to such a scenario. Also, while our definition assumes a single reference file, f_{old} , there could be

several similar files that might be helpful in communicating the contents of f_{new} to the client, as discussed later.

In the case of the file synchronization problem, many currently known protocols [16, 35, 49] consist of a single round of communication, where the client first sends a request with a limited amount of information about f_{old} to the server, and the server then sends an encoding of the current file to the client. In the case of a multiround protocol, a standard model for communication costs based on latency and bandwidth can be employed to measure the cost of the protocol. A simple model commonly used in distributed computing defines the cost (time) for sending a message of length m as $L + m/B$, where L is the latency (delay) and B is the bandwidth of the connection.

There are several other interesting algorithmic problems arising in the context of delta compression and remote file synchronization that we also address. For example, in some cases there is no obvious similar file, and we may have to select the most appropriate reference file(s) from a collection of files. In the case of remote file synchronization, we would often like to estimate file similarity efficiently over a network. Finally, the data to be synchronized may consist of a large number of small records, rather than a few large files, necessitating a somewhat different approach.

13.1.2 Content of This Chapter

In this chapter, we survey techniques, software tools, and applications for delta compression and remote file synchronization. We consider scenarios in networking as well as storage. For simplicity, most of the time, we consider the case of a single reference file, though the case of more than one file is also discussed. We also discuss related problems such as how to select appropriate reference files for delta compression, how to estimate the similarity of two files, and how to reconcile large collections of record-based data.

In Section 13.2, we focus on delta compression, where the sender knows all the similar files that are held by the receiver. In Section 13.3, we survey work on the related, but in many ways quite different, problem of remote file synchronization, where the sender does not have a copy of the files held by the receiver. Finally, Section 13.4 offers some concluding remarks.

13.2 DELTA COMPRESSION

We now focus on the delta compression problem. We first describe some important application scenarios that benefit from delta compression. In Subsection 13.2.2 we give an overview of delta compression approaches, and Subsection 13.2.3 describes in more detail a delta compressor based on the Lempel-Ziv (LZ) compression algorithm. Experimental results for a few delta compressors are given in Subsection 13.2.4. Finally, we discuss the problems of space-constrained delta compression and of choosing good reference files in Subsections 13.2.5 and 13.2.6, respectively.

13.2.1 Applications

As mentioned above, most of the applications of delta compression are aimed at reducing networking or storage costs. We now describe a few of them in more detail.

13.2.1.1 Software Revision Control Systems

As mentioned in the Introduction, delta compression techniques were pioneered in the context of systems used for maintaining the revision history of software projects and other documents [8, 40, 45]. In these systems, multiple, often almost identical, versions of each object must be

stored in order to allow the users to retrieve past versions. For example, the RCS (Revision Control System) package [45] uses the *diff* delta compressor to reduce storage requirements. For more discussion on delta compression in the context of such systems, and an evaluation of different compressors, see the work of Hunt *et al.* [25].

13.2.1.2 Delta Compression at the File System Level

The Xdelta File System (XDFS) of MacDonald [29] aims to provide efficient support for delta compression at the file system level using a delta compressor called *xdelta*. This allows the efficient implementation of revision control systems, as well as some other applications listed here, on top of XDFS.

13.2.1.3 Software Distribution

As described in the example in the Introduction, delta compression techniques are used to generate software patches that can be efficiently transmitted over a network in order to update installed software packages.

13.2.1.4 Exploring File Differences

Techniques from delta compression can be used to visualize differences between different documents. For example, the well-known *diff* utility displays the differences between two files as a set of edit commands, while the *HtmlDiff* and *topblend* tools of Chen *et al.* [14] visualize the difference between two HTML documents.

13.2.1.5 Improving HTTP Performance

Several approaches have been proposed that employ delta compression to improve the latency for web accesses, by exploiting the similarity between current and outdated versions of a web page, and between different pages on the same web site. In particular, Banga *et al.* [6] and Mogul *et al.* [34] propose a scheme called *optimistic delta* in which a caching proxy attempts to hide the latency of server replies by first sending a potentially outdated cached version of a page to the client and then if necessary a small corrective patch once the server replies. In another approach, a client that already has an old version of a page in his cache sends a tag identifying this version to a proxy (or server) as part of the HTTP request; the proxy then sends the delta between the old and the current version to the client [18, 24]. This can significantly decrease the amount of data sent to the client and is thus more appropriate for clients connected via low-bandwidth links such as cellular modems.

It has also been observed that web pages on the same server often have a high degree of similarity (due to common layout and menu structure) that could be exploited with delta compression techniques. In particular, Chan and Woo [13] propose to identify candidate pages that are likely to be good reference files for delta compression by looking for URLs that share a long common prefix with the requested one. Other work [19] proposes a similar idea for dynamic pages, e.g., different stock quotes from a financial site, that share a lot of content.

13.2.1.6 Efficient Web Page Storage

The similarities between different versions of the same page or different pages on the same web site could also be used for increased storage efficiency in large-scale web repositories such as the Internet Archive¹ or the Stanford WebBase [23]. In particular, the Internet Archive aims to preserve multiple versions of each page, but these pages are currently stored without the use of delta compression techniques. A delta-compressed archive could be implemented on top of XDFS [29],

¹ <http://www.archive.org>.

but a specialized implementation with a more optimized delta compressor might be preferable in this case. If we also plan to exploit similarities between different pages, then the problem of selecting appropriate reference pages arises. While the general formulation of this problem, discussed further below, is quite challenging, specialized techniques based on URL prefix matching [13] plus separate detection of mirrors [9] and replicated collections [15] may suffice in practice.

13.2.2 Fundamentals

Recall that in the delta compression problem, we have two files, f_{old} and f_{new} , and the goal is to compute a file f_δ of minimum size, such that one can reconstruct f_{new} from f_{old} and f_δ . Early work on this problem was done within the framework of the *string-to-string correction problem*, defined in [50] as the problem of finding the best sequence of insert, delete, and update operations that transform one string to another. Approaches for solving this problem were based on finding the largest common subsequence of the two strings using dynamic programming and adding all remaining characters to f_{new} explicitly. However, the string-to-string correction problem does not capture the full generality of the delta compression problem as illustrated in the examples given in the previous subsection. For example, in the string-to-string correction problem, it is implicitly assumed that the data common to f_{new} and f_{old} appear in the same order in the two files. Furthermore, the string-to-string correction approach does not account for substrings in f_{old} appearing in f_{new} several times.

To resolve these limitations, Tichy [44] defined the *string-to-string correction problem with block moves*. A *block move* is a triple (p, q, l) such that $f_{old}[p, \dots, p + l - 1] = f_{new}[q, \dots, q + l - 1]$. It represents a non-empty common substring of f_{old} and f_{new} which is of length l . Given f_{old} and f_{new} , the file f_δ can then be constructed as a minimal *covering set* of block moves such that every element $f_{new}[i]$ that also appears in f_{old} is included in exactly one block move. It can be further argued that an f_δ constructed from the longest common subsequence approach mentioned earlier is just a special case of a covering set of block moves. The minimality condition then ensures the superiority of the block-moves approach to the longest common subsequence approach.

The question then arises—how does one construct an optimal f_δ given f_{old} and f_{new} ? Tichy [44] also showed that a greedy algorithm results in a minimal cover set and that an f_δ based on a minimal cover set of block moves can be constructed in linear space and time using suffix trees. Unfortunately, the multiplicative constant in the space complexity makes the approach impractical. A more practical approach uses hash tables with linear space but quadratic time worst case complexity [44].

The block-moves framework described above represented a fundamental shift in the development of delta compression algorithms. While earlier approaches used an edit-based approach—i.e., construct an optimal sequence of edit operations that transform f_{old} into f_{new} , the block-moves algorithms use a *copy-based* approach—i.e., express f_{new} as an optimal sequence of copy operations from f_{old} .

The Lempel–Ziv string compression algorithms [52, 53] popularized in the 1980s yield another natural framework for realizing delta compression techniques based on the copy-based approach. In particular, the LZ77 algorithm can be viewed as a sequence of operations that involve replacing a prefix of the string being encoded with a reference to an identical previously encoded substring. In most practical implementations of LZ77, a greedy approach is used whereby the longest matching prefix found in the previously encoded text is replaced by a copy operation.

Thus, delta compression can be viewed as simply performing LZ77 compression with the file f_{old} representing “previously encoded” text. In fact, nothing prevents us from also including the part of f_{new} which has already been encoded in the search for a longest matching prefix. A few additional changes are required to get a really practical implementation of a LZ77-based delta compression technique. Several such implementations have been designed over the past decade but

the basic framework is the same. The difference lies mostly in the encoding and updating mechanisms employed by each. In the next subsection we describe one such technique in more detail.

13.2.3 LZ77-Based Delta Compressors

The best general-purpose delta compression tools are currently copy-based algorithms based on the Lempel–Ziv [52] approach. Examples of such tools are *vdelta* and its newer variant *vcdiff* [25], the *xdelta* compressor used in *XDFS* [29], and the *zdelta* tool [47].

We now describe the implementation of such a compressor in more detail, using the example of *zdelta*. The *zdelta* tool is based on a modification of the *zlib* compression library of Gailly [21], with some additional ideas inspired by *vdelta*, and anyone familiar with *zlib*, *gzip*, and other Lempel–Ziv-based algorithms should be able to easily follow the description. Essentially, the idea in *zdelta*, also taken in the *vcdiff* (*vdelta*) and *xdelta* algorithms, is to encode the current file by pointing to substrings in the reference file as well as in the already encoded part of the current file.

To identify suitable matches during coding, we maintain two hash tables, one for the reference file, T_{old} , and one for the already coded part of the current file, T_{new} . The table T_{new} is essentially handled the same way as the hash table in *gzip*, where we insert new entries as we traverse and encode f_{new} . The table T_{old} is built beforehand by scanning f_{old} , assuming f_{old} is not too large. When looking for matches, we search in both tables to find the best one. Hashing of a substring is done based on its first three characters, with chaining inside each hash bucket.

Let us assume for the moment that both reference and current files fit into the main memory. Both hash tables are initially empty. The basic steps during encoding are as follows. (Decoding is fairly straightforward given the encoding.)

1. Preprocessing the Reference File:

For $i = 0$ to $\text{len}(f_{old}) - 3$:

- (a) Compute $h_i = h(f_{old}[i, i + 2])$, the hash value of the first three characters starting from position i in f_{old} .
- (b) Insert a pointer to position i into hash bucket h_i of T_{old} .

2. Encoding the Current File:

Initialize pointers p_1, \dots, p_k to zero, say, for $k = 2$.

Set $j = 0$.

While $j \leq \text{len}(f_{new})$:

- (a) Compute $h_j = h(f_{new}[j, j + 2])$, the hash value of the first three characters starting from position j in f_{new} .
- (b) Search hash bucket h_j in both T_{old} and T_{new} to find a “good match,” i.e., a substring in f_{old} or the already encoded part of f_{new} that has a common prefix of maximum length with the string starting at position j of f_{new} .
- (c) Insert a pointer to position j into hash bucket h_j of T_{new} .
- (d) If the match is of length at least 3, encode the position of the match relative to j if the match is in f_{new} and relative to one of the pointers p_i if the match is in f_{old} . If several such matches of the same length were found in (b), choose the one that has the smallest relative distance to position j in f_{new} or to one of the pointers into f_{old} . Also encode the length of the match and which pointer was used as reference. Increase j by the length of the match, and possibly update some of the pointers p_i .
- (e) If there is no match of length at least 3, write out character $f_{new}[j]$ and increase j by 1.

There are a number of additional details to the implementation. First, we can choose a variety of policies for updating the pointers p_i . The motivation for these pointers, as used in *vdelta*, is that in many cases the location of the next match from f_{old} is a short distance from the previous one,

especially when the files are very similar. Thus, by updating one of the pointers to point to the end of the previous match, we hope to very succinctly encode the location of the next match. In general, smart pointer movement policies might lead to additional moderate improvements over the existing tools.

Another important detail concerns the method used to encode the distances, match lengths, pointer information, and characters. Here, *zdelta* uses the Huffman coding facilities provided by *zlib*, while *vdelta* uses a byte-based encoding that is faster but less compact. In contrast, *xdelta* does not try any clever encoding at all, but leaves it up to the user to apply a separate compression tool on the output.

A very important issue is what to do if the hash tables develop very long chains. For T_{new} this can be handled by simply evicting the oldest entries whenever a bucket grows beyond a certain size (as done in *gzip*), but for T_{old} things are more complicated. Note that full buckets can happen for two reasons. First, if f_{old} is very large, then all buckets of T_{old} may become large. This is handled in *vdelta* and *zdelta* by maintaining a “window” of fixed size (say, 32 or 64 kB) into f_{old} . Initially, the window is at the start of the file, and as the encoding proceeds we slide this window through f_{old} according to the positions where the best matches were recently found. A simple heuristic, employed by *zdelta*, uses a weighted average of the most recently used matches to determine when it is time to slide the window by half the window size. In that case, we remove all entries that are outside the new window from T_{old} , and add the new entries that are now inside the window. A more sophisticated scheme, used in *vcdiff*, computes fingerprints on blocks of a certain size (e.g., 1024 bytes) and then chooses a window position in f_{old} that maximizes the similarity with the currently considered area of f_{new} . In general, the problem of how to best move the window though f_{old} is difficult and not really resolved, and the best movement may not be sequential from start to end.

Second, even if f_{old} is small, or during one window position, some buckets can become very large due to a frequently occurring substring. In this case, it is not clear which entries should be evicted: those at the beginning or at the end of the file or current window. This issue is again related to the problem of finding and exploiting the pattern of matches in the reference file, which depends on the relation between the reference file and the current file, and a good solution is not yet apparent.

13.2.4 Some Experimental Results

We now show a few experimental results to give the user a feel for the compression performance of the available tools. In these results, we compare *xdelta*, *vcdiff*, and *zdelta* on two different sets of files:

1. The *gcc* and *emacs* data sets used in the performance study in [25], consisting of versions 2.7.0 and 2.7.1 of *gcc* and 19.28 and 19.29 of *emacs*. The newer versions of *gcc* and *emacs* consist of 1002 and 1286 files, respectively.
2. A set of artificially created files that model the degree of similarity between two files. In particular, we created two completely random files f_0 and f_1 of fixed length, and then performed delta compression between f_0 and another file f_m created by a “blending” procedure that copies text from either f_0 or f_1 according to a simple Markov process. By varying the parameters of the process, we can create a sequence of files f_m with similarity ranging from 0 ($f_m = f_1$) to 1 ($f_m = f_0$) on a non-linear scale.²

² More precisely our process has two states, s_0 , where we copy a character from f_0 , and s_1 , where we copy a character from f_1 , and two parameters, p , the probability of staying in s_0 , and q , the probability of staying in s_1 . In the experiments, we set $q = 0.5$ and vary p from 0 to 1. Clearly, a complete evaluation would have to look at several settings of q to capture different granularities of file changes.

Table 13.1 Compression Results for *gcc* and *emacs* Data Sets (sizes in kilobytes and times in seconds)

	gcc size	gcc time	emacs size	emacs time
Uncompressed	27,288	—	27,326	—
<i>gzip</i>	7,479	24/30	8,191	26/35
<i>xdelta</i>	461	20	2,131	29
<i>vcdiff</i>	289	33	1,821	36
<i>zdelta</i>	250	26/32	1,465	35/42

All runs were performed on a Sun E450 server with two 400-MHz UltraSparc IIe processors and 4 GB of main memory, with the data stored on a 10,000-rpm SCSI disk. We note that only one CPU was used during the runs and that memory consumption was not significant. (We also did multiple runs for each file in the collections and discarded the first one—the main result of this setup is to minimize disk access costs, thus focusing on the CPU costs of the different methods.)

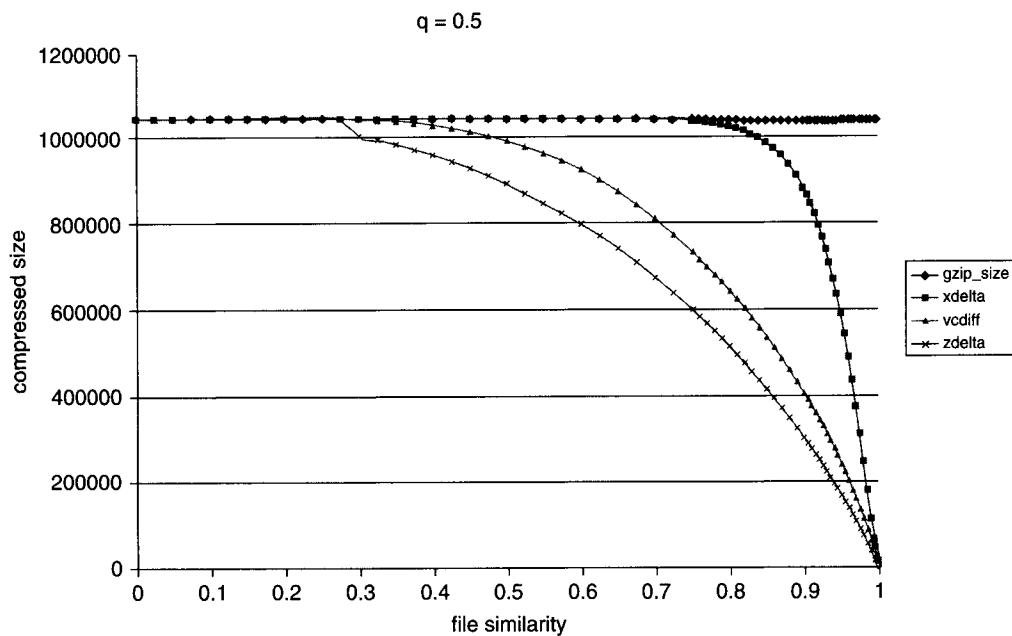
For the *gcc* and *emacs* data sets, the *uncompressed* and *gzip* numbers are with respect to the newer releases. We see from the results that delta compression achieves significant improvements over *gzip* on these files, especially for the very similar *gcc* files (see Table 13.1). Among the delta compressors, *zdelta* gets the best compression ratio, mainly due to the use of Huffman coding instead of byte-based coding. The *xdelta* compressor performs worst in these experiments. As described in [29], *xdelta* aims to separate differencing and compression, and thus a standard compressor such as *gzip* can be applied to the output of *xdelta*. However, in our experiments, subsequent application of *gzip* did not result in any significant improvement on these data sets.

Concerning running times, all three delta compressors are only slightly slower than *gzip*, with *xdelta* coming closest (see Table 13.1). Note that for *gzip* and *zdelta* we report two different numbers that reflect the impact of the input/output method on performance. The first, lower number gives the performance using direct access to files, while the second number is measured using Standard I/O. The number for *vcdiff* is measured using Standard I/O, while *xdelta* uses direct file access. Taking these differences into account, all delta compressors are within at most 20% of the time for *gzip*, even though they must process two sets of files instead of just one as in *gzip*.

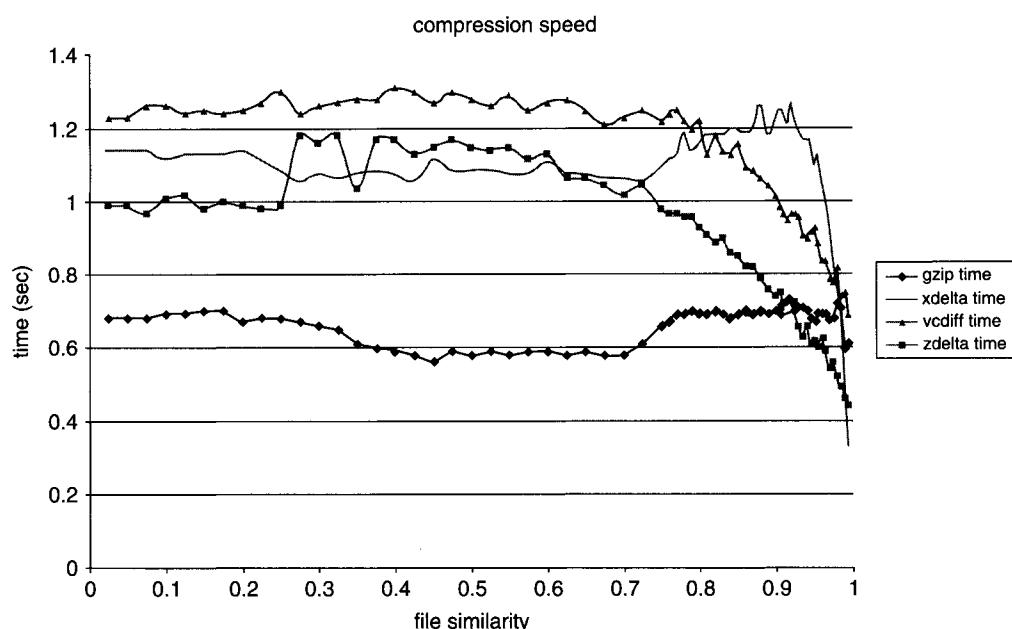
Looking at the results for different file similarities, we see the same ordering (see Fig. 13.1). Not surprisingly, when files are very different, delta compression does not help at all, while for almost identical files, all methods do quite well. However, we see that *vcdiff* and *zdelta* give benefits even for only slightly similar files for which *xdelta* does not improve over *gzip*. (Note that *gzip* itself does not provide any benefits here due to the incompressibility of the files.) We also see that the running times for the delta compressors decrease as file similarity increases (see Fig. 13.2); this is due to the increasing lengths of the matches found in the reference files (which decrease the number of searches in the hash tables). This effect largely explains why the delta compressors are almost as fast as *gzip* on collections with large similarity such as *gcc* and *emacs*; for files with low degrees of similarity, the three delta compressors take about 60 to 100% longer than *gzip*.

13.2.5 Space-Constrained Delta Compression

As described in Subsection 13.2.2, the greedy algorithm of Tichy [44] results in an optimal set of block moves. To show how these block moves are determined and encoded in a practical implementation, in Subsection 13.2.3 we discussed an implementation based on an LZ77-like

**FIGURE 13.1**

Compression versus file similarity (in kilobytes).

**FIGURE 13.2**

Running time versus file similarity (in seconds).

framework. However, these algorithms can give very poor performance when limited memory resources are available to the compressor or decompressor. In [3] Ajtai *et al.* looked at the problem of delta compression under various resource constraints. They first examined delta compression in linear time and constant space, which is relevant when the files f_{old} and f_{new} are too large to fit in memory. A simple solution in this case would be to restrict the search for the longest prefix in f_{old} to the forward direction only. That is, when looking for a match we ignore the part of f_{old} that precedes the end of the substring that was just encoded. However, this results in significantly suboptimal compression when substrings occur in different order in f_{old} and f_{new} .

To alleviate this problem Ajtai *et al.* [3] propose a *correcting one-pass* algorithm which utilizes a buffer that holds all copy commands and performing corrections on these commands later when better matches are found. Two types of corrections are made. *Tail corrections* occur when, after a copy command from a previously unencoded part of f_{old} is inserted, the algorithm attempts to extend the matching string *backward* in both f_{old} and f_{new} . If such matches are found going in a backward direction, there is potential for replacing the previous copy commands by integrating them into the current copy command. The second type of correction, called *general correction*, occurs if a matching substring M is found that is already encoded in f_{new} . In this case, the algorithm tries to determine whether the earlier encoding of M can be further compacted now that M can be encoded by a single copy command. Further, to limit the space consumed by the hash tables that store substring locations, they use a technique called *checkpointing* that restricts the locations of a substring that are inserted into the hash table to a small but carefully selected number. The result of these extensions is a delta compression technique that is practical in terms of time and space complexity, works with arbitrary size files, and yields good compression.

13.2.6 Choosing Reference Files

In some applications, delta compression performance depends heavily on the choice of appropriate reference files. For example, to compress a set of related files, we need to choose for each file one or several reference files that have significant similarity to it; each reference file itself can also be compressed this way, provided that no cycles are created. In the case of a single reference file per compressed file, this problem is equivalent to finding an *optimum branching* in a corresponding directed graph where each edge (i, j) has a weight equal to size of the delta of i with respect to reference file j . This problem can be solved in time quadratic in the number of documents [12, 42], but the approach suffers from two drawbacks: First, the solution may contain very long chains of documents that must be accessed in order to uncompress a particular file. Second, for large collections the quadratic time becomes unacceptable, particularly the cost of computing the appropriate weights of the edges of the directed graph.

If we impose an upper bound on the length of the reference chains, then finding the optimum solution becomes NP Complete [43]. If we allow each file to be compressed using more than one reference file, then this problem can be reduced to a generalization of optimum branching to hypergraphs and has been shown NP Complete even with no bound on the length of chains [2].

Some experiments on small web page collections using minimum branching and several faster heuristics are given in [38], which show significant differences in compression performance between different approaches. For very large collections, general document clustering techniques such as [10, 22, 30], or specialized heuristics such as [9, 13, 15] for the case of web documents, could be applied. In particular, Chan and Woo [13] demonstrate that there is significant benefit in choosing more than one reference page to compress a web page.

One example of long reference chains arises when dealing with many different versions of the same file, such as in a revision control system. In this case, the choice of the reference file that minimizes the delta is usually obvious, but this choice would make retrieval of very old versions

quite expensive.³ Several techniques have been proposed for dealing with this problem [11, 29, 45], by creating a limited number of additional “shortcuts” to older versions.

13.3 REMOTE FILE SYNCHRONIZATION

In this section, we focus on the remote file synchronization problem, i.e., the case where the server does not have access to the reference file. This obviously changes the problem significantly, and the known algorithms for this problem are quite different from those for delta compression. We discuss the two main known approaches for file synchronization: (i) a practical approach based on string fingerprints implemented by the *rsync* algorithm that does not achieve any provable near-optimal bounds and (ii) an approach based on colorings of hypergraphs that achieves provable performance bounds under certain models for file distance, but that seems unsuitable in practice. We also discuss the closely related problems of how to estimate file similarity and how to reconcile sets of records in a database.

We first describe a few application scenarios for file synchronization. In Subsection 13.3.2 we describe the *rsync* algorithms, and Subsection 13.3.3 gives some experimental results comparing *rsync* and delta compressor performance. Subsection 13.3.4 presents some general theoretical bounds, and Subsection 13.3.5 discusses provable results for specific distance measures. Finally, Subsection 13.3.6 discusses how to estimate the distance between two files, while Subsection 13.3.7 looks at the problem of reconciling large sets of data records.

13.3.1 Applications

The applications for file synchronization are similar to those for delta compression. Synchronization is more general in that it does not require knowledge of the reference file; on the other hand, delta compression tends to significantly outperform synchronization in terms of compression ratio. There are several reasons that the server may not have the reference file, such as the space, disk access, or software overhead of maintaining and retrieving old versions of files as references, changes to the file at the client or a third party, or later deletion (eviction) of old versions at the server. Some typical scenarios are shown in the following.

13.3.1.1 Synchronization of User Files

There are a number of software packages such as *rsync* [48, 49] Microsoft’s *ActiveSync*, Puma Technologies’ *IntelliSync*, or Palm’s *HotSync* that allow “synchronization” between desktops, mobile devices, or web-accessible user accounts. In this scenario, files or records can be updated by several different parties, and time stamps may be used to determine which version on which device is the most recent.

We note that there are several challenges for these tools. For data in the form of files, we have the remote file synchronization problem already defined in the Introduction, where we would like to avoid transferring the entire file. For data consisting of large sets of small records, e.g., addresses or appointments on a handheld device, the problem is how to identify those records that have changed without sending an individual fingerprint or time stamp for each record. This problem, modelled as a *set reconciliation* problem in [33], is discussed in Subsection 13.3.7. Many existing packages transfer the entire item if any change has occurred, which is reasonable for small record-based data, but not for larger files. In addition, there is also the general and nontrivial problem of defining the proper semantics for file system synchronization; see [5] for a detailed discussion.

³ Note that these systems often compress older versions with respect to newer ones.

13.3.1.2 *Remote Backup of Massive Data Sets*

Synchronization can be used for remote backup of data sets that may have changed only slightly between backups [48]. In this case, the cost of keeping the old version at the server is usually prohibitive, making delta compression techniques inefficient. (See also [11] for an approach that adapts delta compression to this case.)

13.3.1.3 *Web Access*

File synchronization has also been considered for efficient HTTP transfer between clients and a server or proxy.⁴ The advantage is that the server does not have to keep track of the old versions held by the clients and does not need to fetch such versions from disk upon a request. However, as shown in Subsection 13.3.3, file synchronization techniques achieve significantly worse compression ratios than delta compressors and thus typically provide benefits only for files that are “very similar.” (We are not aware of any study quantifying these advantages and disadvantages for HTTP transfer.)

13.3.1.4 *Distributed and Peer-to-Peer Systems*

Synchronization can be used to update highly distributed data structures, such as routing tables, name services, indexes, or replication tables. A significant amount of recent work has looked at highly distributed and peer-to-peer systems and services. We expect interesting applications of file synchronization and set reconciliation to arise in this context, particularly in peer-to-peer systems where nodes are often unavailable for significant periods of time and thus must update their data structures upon rejoining the system.

13.3.2 The *rsync* Algorithm

We now describe the algorithm employed by the widely used *rsync* file synchronization tool⁵ of Tridgell and MacKerras (see [48, 49]). A similar approach was also proposed by Pyne in a U.S. Patent [39]. For intuition, consider the following simple problem that captures some of the challenges.

Assume that two parties communicate via telephone, with each party holding a copy of a book. Now suppose that the two copies could differ in a few places. How can the two parties find out if the two books are identical or not, and if not where and how they exactly differ, without reading an entire book over the phone? The answer to the first question is simple: by computing a checksum (e.g., MD5) for each book and comparing the two checksums, it can be decided if the books are identical or not. The answer to the second question, however, is more difficult. A first naive approach would partition the book into two blocks, the first and second halves of the book, and then recurse on those blocks where the checksums differ, until the precise locations of the differences are found. However, this approach fails in the simple case where one book contains an additional word at the beginning, thus destroying the alignments of all the block boundaries between the two books. Thus, a more careful approach is needed, although the basic idea of using checksums on blocks is still relevant.⁶

We now describe the refined *rsync* algorithm. Essentially, the idea is to solve the alignment problem by computing blockwise checksums for the reference file and comparing these checksums

⁴ See, e.g., the *rproxy* project at <http://rproxy.samba.org/>.

⁵ Available at <http://rsync.samba.org/>.

⁶ Note that if the books are divided into natural components such as chapters, sections, and subsections, then the alignment problem does not arise. This is similar to the setup described in Subsection 13.3.7 where both files consist of individual records with boundaries known to both parties.

not just with the “corresponding” blockwise checksums of the current file, but with the checksums of all possible positions of blocks in the current file. As a result, the server knows which parts of the current file already exist in the reference file and which new parts need to be communicated to the client. For efficiency reasons, two different checksums are communicated to the server, a fast but unreliable one, and a very reliable one that is more expensive to compute.

1. At the client:

- (a) Partition f_{old} into blocks $B_i = f_{old}[ib, (i + 1)b - 1]$ of some size b to be determined later.
- (b) For each block B_i , compute two checksums, $u_i = h_u(B_i)$ and $r_i = h_r(B_i)$, and communicate them to the server. Here h_u is the unreliable but fast checksum function, and h_r is the reliable but expensive checksum function.

2. At the server:

- (a) For each pair of received checksums (u_i, r_i) , insert an entry (u_i, r_i, i) into a dictionary data structure, using u_i as key value.
- (b) Perform a pass through f_{new} , starting at position $j = 0$ and involving the following steps:
 - (i) Compute the unreliable checksum $h_u(f_{new}[j, j + b - 1])$ on the block starting at j .
 - (ii) Check the dictionary for any block with a matching unreliable checksum.
 - (iii) If found, and if the reliable checksums also match, transmit a pointer to the index of the matching block in f_{old} to the client, advance j by b positions, and continue.
 - (iv) If none found, or if the reliable checksums did not match, transmit the symbol $f_{new}[j]$ to the client, advance j by one position, and continue.

3. At the client:

- (a) Use the incoming stream of data and pointers to blocks in f_{old} to reconstruct f_{new} .

Thus, the fast and unreliable checksum is used to find likely matches, and the reliable checksum is then used to verify the validity of the match.⁷ The reliable checksum is implemented using MD4 (128 bits). The unreliable checksum is implemented as a 32-bit “rolling checksum” that allows efficient sliding of the block boundaries by one character; i.e., the checksum for $f[j + 1, j + b]$ can be computed in constant time from $f[j, j + b - 1]$.

Clearly, the choice of a good block size is critical to the performance of the algorithm. Unfortunately, the best choice is highly dependent on the degree of similarity between the two files—the more similar the files are, the larger the block size we can choose. Moreover, the location of changes in the file is also important. If a single character is changed in each block of f_{old} , then no match will be found by the server and *rsync* will be completely ineffective; on the other hand, if all changes are clustered in a few areas of the file, *rsync* will do very well even with a large block size. Given these observations, some basic performance bounds based on block size and number and size of file modifications can be shown. However, *rsync* does not have any good performance bounds with respect to common file distance metrics such as edit distance [37].

Of course, in general the optimal block size changes even within a file. In practice, *rsync* starts out with a block size of several hundred bytes and uses heuristics to adapt the block size later. Another optimization in *rsync* allows the server to compress all transmitted symbols for unmatched parts of the file using the LZ compression algorithm; this gives significant additional benefits in many situations as shown in the following.

⁷ In addition, a checksum on the entire file is used to detect the (extremely unlikely) failure of the reliable checksum, in which case the entire procedure is repeated with a different choice of hash functions.

13.3.3 Some Experimental Results for *rsync*

We now provide some experimental results to give the reader an idea about the performance of *rsync* in comparison to delta compression techniques. The results use the *gcc* and *emacs* data sets from Subsection 13.2.4. We report five different numbers for *rsync*: the amount of data sent from client to server (request), the amount of data sent from server to client (reply), the amount sent from server to client with compression option switched on (reply compressed), and the total for both directions in uncompressed (total) and compressed (total compressed) form.

We observe that without compression option, *rsync* does worse than *gzip* on the *emacs* set (see Table 13.2). However, once we add compression for the reply message, *rsync* does significantly better than *gzip*, although it is still a factor of 3 to 4 from the best delta compressor. We also compared *rsync* on the artificial data sets from Subsection 13.2.4; due to the fine distribution of file changes for $q = 0.5$, *rsync* only achieves any benefits at all for p very close to 1. We note that *rsync* is typically applied in situations where the two files are very similar, and hence these numbers may look overly pessimistic. Clearly, *rsync* provides benefits to many people who use it on a daily basis.

In Table 13.3, we see how the performance of *rsync* varies as we decrease the block size used. The size of the server reply decreases as a smaller block size is used, since this allows a finer granularity of matches. Of course, the size of the request message increases, since more hash values need to be transmitted, and eventually this overcomes the savings for the reply (especially in the compressed case since the hash values in the request are incompressible).

In summary, there still exists a gap between delta compression and remote file synchronization techniques in terms of performance; we believe that this indicates room for significant

Table 13.2 Compression Results for *gcc* and *emacs* Data Sets (in kilobytes)

	gcc	emacs
Uncompressed	27,288	27,326
<i>gzip</i>	7,479	8,191
<i>xdelta</i>	461	2,131
<i>vcdiff</i>	289	1,821
<i>zdelta</i>	250	1,465
<i>rsync</i> request	180	227
<i>rsync</i> reply	2,445	12,528
<i>rsync</i> reply compressed	695	4,200
<i>rsync</i> total	2,626	12,756
<i>rsync</i> total compressed	876	4,428

Table 13.3 Compression Results for *emacs* with Different Block Sizes (in kilobytes)

	700	500	300	200	100	80
<i>rsync</i> request	227	301	472	686	1328	1649
<i>rsync</i> reply	12,528	11,673	10,504	9,603	8433	8161
<i>rsync</i> reply compressed	4,200	3,939	3,580	3,283	2842	2711
<i>rsync</i> total	12,756	11,974	10,976	10,290	9762	9810
<i>rsync</i> total compressed	4,428	4,241	4,053	3,970	4170	4360

improvements. One promising approach uses multiple roundtrips between client and sever, e.g., to determine the best block size or to recursively split blocks; see [16, 20, 37] for such methods. Recent experimental results by Orlitsky and Viswanathan [37] show improvements over *rsync* using such a multiround protocol.⁸

13.3.4 Theoretical Results

In addition to the heuristic solution given by *rsync*, a number of theoretical approaches have been studied that achieve provable bounds with respect to certain formal measures of file similarity. Many definitions of file similarity have been studied, e.g., *Hamming distance*, *edit distance*, or measures related to the compression performance of the Lempel–Ziv compression algorithm [16]. Results also depend on the number of messages exchanged between the two parties (e.g., in *rsync*, two messages are exchanged). In this subsection, we describe the basic ideas underlying these approaches and give an overview of known results, with emphasis on a few fundamental bounds presented by Orlitsky [35]. See [17, 28] and the references in [35] for some earlier results on this problem. The results in [35] are stated for a very general framework of pairs of random variables; in the following we give a slightly simplified presentation for the case of correlated (similar) files.

13.3.4.1 Distance Measures

We first discuss the issue of *distance measures*, which formalize the notion of file similarity. Note that Hamming distance, one of the most widely studied measures in coding theory, is not very appropriate in our scenario, since a single insertion of a character at the beginning of a file would result in a very large distance between the old and new files, while we expect a reasonable algorithm to be able to synchronize these two files with only a few bytes of communication. In the *edit distance* measure, we count the number of single-character change, insertion and deletion operations needed to transform one file into another, while more generalized notions of edit distance also allow for deletions and moves of blocks of data or may assign different weights to the operations. Finally, another family of distance measures is based on the number of operations needed to construct one file by copying blocks over from the other file and by inserting single characters; an example is the LZ measure proposed in [16]. As already discussed in Section 13.2, a realistic measure should allow for moves and copies of large blocks; however, from a theoretical point of view allowing such powerful operations makes things more complicated.

There are a few properties of distance functions that we need to discuss. A distance measure is called *symmetric* if the distance from f_0 to f_1 is the same as the distance from f_1 to f_0 . This property is satisfied by Hamming and edit distance, but is not true for certain generalized forms of edit distance and copy-based measures. A distance measure d is a metric if (a) it is symmetric, (b) $d(x, y) \geq 0$ for all x, y , (c) $d(x, y) = 0$ iff $x = y$, and (d) it observes the Triangle Inequality.

13.3.4.2 Balanced Pairs

Now assume that as part of the input, we are given upper bounds $d_{new} \geq d(f_{old}, f_{new})$ and $d_{old} \geq d(f_{new}, f_{old})$ on the distances between the two files. We define $N_k(f)$, the k -neighborhood of a file f , as the set of all files f' such that $d(f', f) \leq k$. Thus, given the upper bounds on the distances, the server holding f_{new} knows that f_{old} is one of the files in $N_{d_{new}}(f_{new})$, while the client holding f_{old} knows that f_{new} is in $N_{d_{old}}(f_{old})$. We refer to the size of $N_{d_{new}}(f_{new})$ (resp. $N_{d_{old}}(f_{old})$) as the *ambiguity* of f_{new} (resp. f_{old}). We assume that both parties know *a priori* upper

⁸ Note that these roundtrips are not incurred on a per-file basis, since we can handle many files at the same time. Thus, latency due to additional roundtrips is not a problem in many situations, although the amount of “state” that has to be maintained between roundtrips may be more of an issue.

bounds on both ambiguities, referred to as the *maximum ambiguity* of f_{new} (resp. f_{old}), written $\text{mamb}(f_{\text{new}})$ (resp. $\text{mamb}(f_{\text{old}})$). (In the presentation in [35], these bounds are implied by the given random distribution; in our context, we can assume that both parties compute estimates of their ambiguities beforehand based on file lengths and estimates of file distances.⁹

We say that a pair of files f_{new} and f_{old} is balanced if $\text{mamb}(f_{\text{new}}) = \text{mamb}(f_{\text{old}})$. Note that this property may depend on the choice of the two files as well as the distance measure (and possibly also the value of k). In the case of Hamming distance, all pairs of equal size are balanced, while for many edit and copy-based measures this is not true. In general, the more interesting measures are neither metrics nor generate balanced pairs. However, some results can be shown for distance measures that can be approximated by metrics or in cases where the ambiguities are not too different.

13.3.4.3 Results

We now describe a few fundamental results given by Orlitsky in [35, 36]. We assume that both parties *a priori* have some upper bounds on the distances between the two files. The goal is to limit the number of bits communicated in the worst case for a given number of roundtrips, with unbounded computational power available at the two parties (as commonly assumed in the formal study of communication complexity [27]). In the following, all logarithms are with base 2.

Theorem 13.1. *At least $\lceil \log(\text{mamb}(f_{\text{old}})) \rceil$ bits must be communicated from server to client by any protocol that works correctly on all pairs f_{new} and f_{old} with $d(f_{\text{new}}, f_{\text{old}}) \leq d_{\text{old}}$, independent of the number of messages exchanged and even if the server knows f_{old} [36].*

This first result follows directly from the fact that the client needs to be able to distinguish between all $\text{mamb}(f_{\text{old}})$ possible files f_{new} that satisfy $d(f_{\text{new}}, f_{\text{old}}) \leq d_{\text{old}}$. (Note that the result is independent of d_{new} .) Interestingly, if we send only a single message from server to client, we can match this result up to a factor of 2 in the case of a balanced pair, as shown in the following result.

Theorem 13.2. *There is a protocol that sends a single message of at most $\log(\text{mamb}(f_{\text{old}})) + \log(\text{mamb}(f_{\text{new}})) + 1$ bits from server to client and that works on all f_{new} and f_{old} with $d(f_{\text{new}}, f_{\text{old}}) \leq d_{\text{old}}$ and $d(f_{\text{old}}, f_{\text{new}}) \leq d_{\text{new}}$ [35].*

The result is obtained by considering the *characteristic hypergraph* for the problem, first defined by Witsenhausen [51] and obtained by adding a vertex for each file $f_{\text{new}} \in \Sigma^*$ and for each $f_{\text{old}} \in \Sigma^*$ a hyperedge $E(f_{\text{old}}) = \{f_{\text{new}} \mid d(f_{\text{new}}, f_{\text{old}}) \leq d_{\text{old}} \text{ and } d(f_{\text{old}}, f_{\text{new}}) \leq d_{\text{new}}\}$. Since each vertex is adjacent to at most $\text{mamb}(f_{\text{new}})$ edges and each edge contains at most $\text{mamb}(f_{\text{old}})$ vertices, the chromatic number of the hypergraph is at most $\text{mamb}(f_{\text{old}}) \cdot \text{mamb}(f_{\text{new}})$. If the server sends to the client the color of f_{new} , using $\lceil \log((\text{mamb}(f_{\text{old}}) \cdot \text{mamb}(f_{\text{new}}))) \rceil \leq \log(\text{mamb}(f_{\text{old}})) + \log(\text{mamb}(f_{\text{new}})) + 1$ bits, then the client can reconstruct f_{new} .

As shown by Kahn and Orlitsky (see [35]), this result is almost tight for single-message protocols. However, if we allow more than a single message, much better results can be obtained, as briefly outlined in the following:

Theorem 13.3. *There exist protocols that work on all f_{new} and f_{old} with $d(f_{\text{new}}, f_{\text{old}}) \leq d_{\text{old}}$ and $d(f_{\text{old}}, f_{\text{new}}) \leq d_{\text{new}}$ and that achieve the following bounds [35]:*

- (a) *at most $2 \log(\text{mamb}(f_{\text{old}})) + \log \log(\max\{\text{mamb}(f_{\text{new}}), \text{mamb}(f_{\text{old}})\}) + 4$ bits with two messages exchanged between client and server,*

⁹ Note that explicitly transmitting the precise values of the ambiguities would require about the same amount of data transfer as the file reconciliation problem itself, as implied by the bounds below.

- (b) at most $\log(mamb(f_{old})) + 3 \log \log(\max\{mamb(f_{new}), mamb(f_{old})\}) + 11$ bits with three messages exchanged between client and server, and
- (c) at most $\log(mamb(f_{old})) + 4 \log \log(mamb(f_{old}))$ bits with four messages exchanged between client and server.

The bound for two messages is based on a fairly simple and elegant construction of a family of perfect hash functions using the Lovasz Local Lemma [4]. The bound itself improves on the one-message bound above only for very unbalanced pairs, i.e., when $mamb(f_{old})$ is much smaller than $mamb(f_{new})$, but the result is the main building block for the three-message case. The three-message result is almost optimal for approximately balanced pairs, but not for very unbalanced pairs. This problem is resolved by the four-message protocol, which is in fact independent of $mamb(f_{new})$ and depends only on $mamb(f_{old})$. Thus, at most four messages suffice in principle to get a bit-optimal protocol, up to lower order terms.

While these results are very important in characterizing the fundamental complexity of the remote file synchronization problem, they suffer from three main limitations. First, the protocols do not seem to imply any efficiently implementable algorithms, since the client would have to check a large number of possible files f_{new} in order to find the one that has the color or hash value generated by the protocol. Second, many of the results rely on the existence of balanced pairs, and third, it is not clear what bounds are implied for interesting distance measures, which as discussed are rarely balanced or symmetric. This last issue is discussed in the next subsection.

13.3.5 Results for Particular Distance Measures

We now discuss bounds on communication that can be achieved for particular distance measures, focusing on results from [16, 35, 37].

In general, protocols for a distance measure typically consist of a first phase that uses a distance estimation technique to get an upper bound on the distances between the two files (discussed in the next subsection) and a second phase based, e.g., on one of the protocols of Orlitsky [35], that synchronizes the two files. In order to show bounds for the second phase relative to the file distance under a particular metric, we need to (i) investigate whether or not the files can be assumed to be balanced under the distance measure and select the appropriate protocol, and (ii) bound the maximum ambiguities based on the properties of the distance measures, the upper bounds on the distances, and the lengths of the files.

This approach is taken in [16, 35] to show bounds for a few distance measures. In particular, [16] derives the following bounds based on file lengths and distances:

- $O(\log(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the Hamming distance measure, with two roundtrips;
- $O(\log^3(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the edit distance measure and the LZ measure introduced in [16], with two roundtrips; and
- $O(\log(|f_{new}|) \cdot d(f_{new}, f_{old}))$ bits under the edit distance measure and the LZ measure, with $\log^2(|f_{new}|) + 1$ roundtrips.

In these results, all except the last roundtrip are used for distance estimation; the last bound is based on a more precise distance estimation. With the exception of the Hamming distance, where efficient coding techniques are known [1], these results are not practical due to the above-mentioned problem of efficiently decoding at the recipient.

Some more practical protocols are also given in [16, 37, 41]. The protocols in [16, 37] use a hierarchical partitioning approach, resulting in a logarithmic number of roundtrips but avoiding the decoding problems of the other approaches. Some experimental results in [37] show significant improvements for some data sets as compared to *rsync*, with a protocol that gives provable bounds for a variant of edit distance.

13.3.6 Estimating File Distances

As mentioned, many protocols require *a priori* knowledge of upper bounds on the distances between the two files. We discuss protocols for estimating these distances. We note that we could apply the sampling techniques in [10, 30] to construct fingerprints of the files that could be efficiently transmitted (see also [22] for the problem of finding similar files in larger collections). While these techniques may work well in practice to decide how similar two files are, they are not designed with any of the common distance measures in mind, but are based on the idea of estimating the number of common substrings of a given length.

Good distance estimation techniques for some specific metrics are described in [16]. These techniques consist of a single roundtrip in which a total of $O(\log n)$ bits are sent to exchange fingerprints obtained by appropriate sampling techniques. In particular, [16] shows that the LZ measure, which for the purpose of compression captures a lot of the notion of file similarity, as well as the edit distance, can be approximated through a slightly modified LZ-like measure that is in fact a metric and that this metric itself can be converted into Hamming distance.

13.3.7 Reconciling Database Records and File Systems

In the remote file synchronization problem, we assume that we have already identified two corresponding files, f_{new} and f_{old} , that need to be synchronized, and we would like to avoid transmitting an entire file to do so. However, in many scenarios we have a large number of items (files in a file system or records in a database), only a few of which have been changed. In this case, we would like to identify those items that have been changed without transmitting a separate fingerprint or time stamp for each one. Once we have identified these items, we can then synchronize them using the remote file synchronization techniques presented earlier or by transmitting the entire item in the case of small database records. In the following, we discuss the problems arising in this scenario, with emphasis on a recent approach described in [26, 33, 46]. Some earlier work on reconciliation of record-based data appeared in [1, 7, 31, 32].

Consider the case, assumed in [46], of a handheld device using the Palm Hotsync program to synchronize its database of addresses or appointments with a desktop device. If the handheld was last synchronized with the same desktop, then the Palm Hotsync software can use auxiliary logging information to efficiently identify items that need to be synchronized. However, in the general case where the two parties have not recently synchronized and where both may have added or deleted data, simple logging information will not help, and the software transmits all records.

Assume that we compute a fingerprint of small, fixed size (e.g., MD4 with 128 bits) for each object and that S_{new} is the set of fingerprints (integers) held by the server and S_{old} is the set of fingerprints held by the client.¹⁰ Then the *set reconciliation problem* is the problem of determining the differences $S_{old} - S_{new}$ and $S_{new} - S_{old}$ of the two sets at the client. This then allows the client to decide which records it needs to request from and upload to the server. (Of course, in other cases, each party might want to obtain one of the differences.)

The reader might already observe that this scenario is now quite similar to that encountered in the context of error correcting codes (erasure codes) and this observation is used by Minsky *et al.* [33] to apply techniques from coding theory to the problem. In particular, one solution based on Reed–Solomon codes comes within a factor 2 of the information-theoretic lower bound, while a solution based on the interpolation of characteristic polynomials comes very close to optimal. The protocols in [33, 46] assume that an upper bound on the number of differences between the

¹⁰ Note that if the records are very short, e.g., a few bytes, then we can directly perform reconciliation of the data without using fingerprints.

two sets is known, which can be obtained either by guessing as described in [46] or possibly by using known techniques for estimating set intersection sizes in [10, 30].

Experiments in [46] on a Palm OS handheld device demonstrate significant benefits over the Palm Hotsync approach in many cases. One critical issue is still the amount of computation required, which depends heavily on the number of differences between the two sets.

We note that we could also place S_{new} into a file f_{new} , and S_{old} into a file f_{old} , in sorted order and then apply *rsync* to these files (or even to a concatenation of the original files or records). However, this would not achieve the best possible bounds since set reconciliation is really an easier problem than file synchronization due to the assumption of sets with known record boundaries.

13.4 CONCLUSIONS AND OPEN PROBLEMS

In this chapter, we have described techniques, tools, and applications for delta compression and remote file synchronization problems. We believe that the importance of these problems will increase as computing becomes more and more network-centric, with millions of applications distributing, sharing, and modifying files on a global basis.

In the case of delta compression, existing tools already achieve fairly good compression ratios, and it will be difficult to significantly improve upon these results, although more modest improvements in terms of speed and compression are still possible. Recall that the best delta algorithms are currently based on Lempel–Ziv-type algorithms. In the context of (non-delta) compression, such algorithms, while not providing the absolute best compression for particular types of data, are still considered competitive for general-purpose compression. We would expect a similar situation for delta compression, with major improvements possible only for special types of data. On the other hand, a lot of work still remains to be done on how to best use delta compression techniques, e.g., how to cluster files and identify good reference files, and what additional applications might benefit from delta compression techniques.

For remote file synchronization techniques, on the other hand, there still seems to be a significant gap in compression performance between the currently available tools and the theoretical limits. It is an important question whether we can modify the theoretical approaches from communication complexity into efficiently implementable algorithms with provably good performance. A more realistic goal would be to engineer the approach in *rsync* in order to narrow the gap in performance between remote file synchronization and delta compression.

ACKNOWLEDGMENTS

Thanks are extended to Dimitre Trendafilov for his work on implementing the *zdelta* compressor and to Dimitre Trendafilov and Patrick Noel for providing the experimental data. This work was supported by NSF CAREER Award NSF CCR-0093400 and by Intel Corp.

13.5 REFERENCES

1. Abdel-Ghaffar, K., and A. El Abbadi, 1994. An optimal strategy for comparing file copies. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 1, pp. 87–93, January 1994.
2. Adler, M., and M. Mitzenmacher, 2001. Towards compressing web graphs. In *Proceedings of the IEEE Data Compression Conference (DCC)*, March 2001.
3. Ajtai, M., et al. 2000. Compactly Encoding Unstructured Inputs with Differential Compression. IBM Research Report RJ 10187, September 2000.

4. Alon, N., and J. Spencer, 1992. *The Probabilistic Method*. 1992, John Wiley & Sons, NY.
5. Balasubramaniam, S., and B. Pierce, 1998. What is a file synchronizer? In *Proceedings of the ACM/IEEE MOBICOM'98 Conference*, pp. 98–108, October 1998.
6. Banga, G., F. Douglis, and M. Rabinovich, 1997. Optimistic deltas for WWW latency reduction. In *1997 USENIX Annual Technical Conference, Anaheim, CA*, pp. 289–303, January 1997.
7. Barbara, D., and R. Lipton, 1991. A class of randomized strategies for low-cost comparison of file copies. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, pp. 160–170, April 1991.
8. Berliner, B., 1990. CVS II: Parallelizing software development. In *Proceedings of the Winter 1990 USENIX Conference*, pp. 341–352, January 1990.
9. Bharat, K., and A. Broder, 1999. Mirror, mirror on the web: A study of host pairs with replicated content. In *Proceedings of the 8th International World Wide Web Conference*, May 1999.
10. Broder, A., 1997. On the resemblance and containment of documents. In *Compression and Complexity of Sequences (SEQUENCES'97)*, pp. 21–29. IEEE Comput. Soc., Los Alamitos, CA.
11. Burns, R., and D. Long, 1997. Efficient distributed backup with delta compression. In *Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*.
12. Camerini, P., L. Fratta, and F. Maffioli, 1979. A note on finding optimum branchings. *Networks*, Vol. 9, pp. 309–312.
13. Chan, M., and T. Woo, 1999. Cache-based compaction: A new technique for optimizing web transfer. In *Proceedings of INFOCOM'99*, March 1999.
14. Chen, Y., F. Douglis, H. Huang, and K. Vo, 2000. Topblend: An efficient implementation of HtmlDiff in Java. In *Proceedings of the WebNet2000 Conference*, October 2000.
15. Cho, J., N. Shivakumar, and H. Garcia-Molina, 2000. Finding replicated web collections. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 355–366, May 2000.
16. Cormode, G., M. Paterson, S. Sahinalp, and U. Vishkin, 2000. Communication complexity of document exchange. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, January 2000.
17. Costa, M., 1983. Writing on dirty paper. *IEEE Transactions on Information Theory*, pp. 439–441, May 1983.
18. Delco, M., and M. Ionescu, 2000. xProxy: A transparent caching and delta transfer system for web objects. Unpublished manuscript, May 2000.
19. Douglis, F., A. Haro, and M. Rabinovich, 1997. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, December 1997.
20. Evgimievski, A., 1998. A probabilistic algorithm for updating files over a communication link. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 300–305, January 1998.
21. Gailly, J., zlib compression library. Available at: <http://www.gzip.org/zlib/>.
22. Haveliwala, T. H., A. Gionis, and P. Indyk, 2000. Scalable techniques for clustering the web. In *Proceedings of the WebDB Workshop*, Dallas, TX, May 2000.
23. Hirai, J., S. Raghavan, H. Garcia-Molina, and A. Paepcke, 2000. WebBase: A repository of web pages. In *Proceedings of the 9th International World Wide Web Conference*, May 2000.
24. Housel, B., and D. Lindquist, 1996. WebExpress: A system for optimizaing web browsing in a wireless environment. In *Proceedings of the 2nd ACM Conference on Mobile Computing and Networking*, pp. 108–116, November 1996.
25. Hunt, J., K. P. Vo, and W. Tichy, 1998. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, Vol. 7.
26. Karpovsky, M., L. Levitin, and A. Trachtenberg, 2001. Data verification and reconciliation with generalized error-control codes. In *39th Annual Allerton Conference on Communication, Control, and Computing*.
27. Kushilevitz, E., and N. Nisan, 1997. *Communication Complexity*. Cambridge Univ. Press, Cambridge, UK.
28. Levenshtein, V. I., 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, Vol. 10, No. 8, pp. 707–710. February 1966.
29. MacDonald, J., 2000. *File System Support for Delta Compression*, M.S. thesis, University of California at Berkeley, May 2000.

30. Manber, U., and S. Wu, 1994. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the 1994 Winter USENIX Conference*, pp. 23–32, January 1994.
31. Metzner, J., 1983. A parity structure for large remotely located replicated data files. *IEEE Transactions on Computers*, Vol. 32, No. 8, pp. 727–730, August 1983.
32. Metzner, J., 1991. Efficient replicated remote file comparison. *IEEE Transactions on Computers*, Vol. 40, No. 5, pp. 651–659, May 1991.
33. Minsky, Y., A. Trachtenberg, and R. Zippel, 2000. Set Reconciliation with Almost Optimal Communication Complexity, Technical Report TR2000-1813, Cornell University, Ithaca, NY.
34. Mogul, J. C., F. Douglis, A. Feldmann, and B. Krishnamurthy, 1997. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference*, pp. 181–196.
35. Orlitsky, A., 1993. Interactive communication of balanced distributions and of correlated files. *SIAM Journal of Discrete Math.*, Vol. 6, No. 4, pp. 548–564.
36. Orlitsky, A., 1991. Worst-case interactive communication. II. Two messages are not optimal. *IEEE Transactions on Information Theory*, Vol. 37, No. 4, pp. 995–1005, July 1991.
37. Orlitsky, A. and K. Viswanathan, 2001. Practical algorithms for interactive communication. In *IEEE Int. Symp. on Information Theory*, June 2001.
38. Ouyang, Z., N. Memon, T. Suel, and D. Trendafilov, 2002. Cluster-based delta compression for collections of web pages. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering (WISE)*, December 2002.
39. Pyne, C., 1995. Remote File Transfer Method and Apparatus. U.S. Patent 5446888.
40. Rochkind, M., 1975. The source code control system. *IEEE Transactions on Software Engineering*, Vol. 1, pp. 364–370, December 1975.
41. Schwarz, T., R. Bowdidge, and W. Burkhard, 1990. Low cost comparison of file copies. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 196–202.
42. Tarjan, R., 1977. Finding optimum branchings. *Networks*, Vol. 7, pp. 25–35.
43. Tate, S., 1997. Band ordering in lossless compression of multispectral images. *IEEE Transactions on Computers*, Vol. 46, No. 45, pp. 211–320.
44. Tichy, W., 1984. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, pp. 309–321, November 1984.
45. Tichy, W., 1985. RCS: A system for version control. *Software—Practice and Experience*, Vol. 15, pp. 637–654, July 1985.
46. Trachtenberg, A., D. Starobinski, and S. Agarwal, 2001. Fast PDA Synchronization Using Characteristic Polynomial Interpolation, Technical Report BU-2001-03, Department of Electrical and Computer Engineering. Boston University.
47. Trendafilov, D., N. Memon, and T. Suel, 2002. zdelta: A simple delta compression tool. Technical Report, CIS Department, Polytechnic University, Brooklyn, NY, June 2002.
48. Tridgell, A., 2000. *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Australian National University, Canberra, Australia, April 2000.
49. Tridgell, A., and P. MacKerras, 1996. The rsync Algorithm, Technical Report TR-CS-96-05, Australian National University, Canberra, Australia, June 1996.
50. Wagner, R. A., and M. J. Fisher, 1973. The string-to-string correction problem. *Journal of the ACM*, Vol. 21, No. 1, pp. 168–173, January 1973.
51. Witsenhausen, H., 1976. The zero-error side information problem and chromatic numbers. *IEEE Transactions on Information Theory*, Vol. 22, No. 5, pp. 592–593, September 1976.
52. Ziv, J., and A. Lempel, 1977. A universal algorithm for data compression. *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337–343.
53. Ziv, J., and A. Lempel, 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Vol. 24, No. 5, pp. 530–536.

This Page Intentionally Left Blank

Compression of Unicode Files

PETER FENWICK

14.1 INTRODUCTION

Since the middle 1960s most computers have handled characters using either the 7-bit ASCII or 8-bit EBCDIC character sets and representations. These codes were generally satisfactory for English, less satisfactory for other European languages based on the Roman alphabet, and totally unsuitable for East Asian languages. Various extensions filled the unused 128 characters of the “8-bit” ASCII set or escaped into alternative alphabets using the ASCII “SO—Shift Out” and “SI—Shift In” codes. But there were many such standards and they were generally messy and incompatible.

Unicode [1] represents a concerted effort to develop a unified representation for all known alphabets and ideographic systems. The “canonical” Unicode representation is the 16-bit UCS-2. Other Unicode representations are the UTF-8 recoding, which allows ASCII characters to be represented in 8 bits, but expands others to 2 or 3 bytes and is often used as a distribution format, and UTF-7, which uses 7-bit codes for transmission of Unicode over e-mail and similar systems. Unicode is used in most modern operating systems and programming languages and is a very important development with possible repercussions in areas such as data compressibility.

Because conventional lossless compressors work in octet or byte units while Unicode characters are often represented by several bytes, there is a reasonable suspicion that Unicode compression might differ from that of more conventional files. After a brief overview of Unicode, this chapter will report some work on the compression of UCS-2 and UTF-8 data and especially its behavior with different compressors.

14.2 UNICODE CHARACTER CODINGS

UCS-2 is based on “blocks” of either 128 or 256 characters, with blocks generally allocated to a single language or other special usage. Some examples are shown in Table 14.1. The Unicode Standard lists all of the codes, together with many comments on the design and usage of Unicode

Table 14.1 Examples of Unicode Block Allocation

Code Range	Name	Code Range	Name
U+0000–U+007F	C0 Controls & Basic Latin	U+0F00–U+0FBF	Tibetan
U+0080–U+00FF	C1 Controls & Latin-1 Supplement	U+10A0–U+10FF	Georgian
U+0100–U+017F	Latin Extended-A	U+20D0–U+20FF	Combining Diacritical Marks
U+0370–U+03FF	Greek	U+2200–U+22FF	Mathematical Operators
U+0400–U+04FF	Cyrillic	U+2700–U+27BF	Dingbats
U+0530–U+058F	Armenian	U+3000–U+303F	CJK Symbols and Punctuation
U+0590–U+05FF	Hebrew	U+3040–U+309F	Hiragana
U+0600–U+06FF	Arabic	U+30A0–U+30FF	Katakana
U+0980–U+09FF	Bengali	U+3200–U+32FF	Enclosed CJK Letters and Months
U+0A80–U+0AFF	Gujarati	U+3300–U+33FF	CJK Compatibility
U+0B80–U+0BFF	Tamil	U+4E00–U+9FA5	CJK Ideographs
U+0E00–U+0E7F	Thai	U+AC00–U+D7A3	Hangul Syllables

Table 14.2 Comparison of ASCII Data “Promoted” to Big- and Little-Endian UCS-2

	Byte Order Mark		c	o	d	e
Big-endian	FE	FF	00	63	00	6F
Little-endian	FF	FE	63	00	6F	00
					64	00
					65	00

and conventions for each language; for now it is enough to recognize that its canonical representation, UCS-2, is a 16-bit code, generally structured according to the language or alphabet. The coding of a UCS-2 character is usually written U+wxyz, where the initial U+ signals a hexadecimal code and the 4 hexadecimal digits wxyz give the 16 bits of representation.

14.2.1 Big-endian versus Little-endian

A byte order mark U+FEFF may be prefixed to the Unicode file to resolve any problems of Byte order if files are moved between “big-endian” and “little-endian” computers. A big-endian file will start with the two bytes FE and FF, while a little-endian file will start with FF FE. Table 14.2 shows the bytes of a file with the letters “code” in the two forms.

14.2.2 UTF-8 Coding

While the 16-bit UCS-2 representation is the preferred representation within a computer, transmission often uses sequences of 8-bit bytes or octets according to the UTF-8 standard (UCS Transmission Format 8-bit) as in Table 14.3. Each letter in this table represents a single bit.

A standard ASCII character is emitted “as is” in UTF-8 with a high-order 0 bit. Larger values are broken into 6-bit groups, from the least significant bit. Each group except the most significant is prefixed by the bits “10” and emitted as a byte. The first byte starts with as many 1’s as there are bytes in the code, followed by a 0 (a unary code). While UTF-8 can handle 32-bit characters, only 2- and 3-byte codes are used for UCS-2 characters.

Table 14.3 UCS-2 and UTF-8 Coding

Data Bits	Input Bit Pattern (UCS-2)	Coding into Successive Bytes (UTF-8)	
7	0000 0000 0abc defg	0abc	defg
11	0000 0abc defg hijk	110a	bcde 10fg hijk
16	abcd efgh ijk l mnop	1110 abcd	10ef ghij 10kl mnop

A text string in UCS-2 may, when converted to UTF-8,

1. contract if it contains mostly ASCII characters,
2. remain essentially the same size if it is mostly Cyrillic, Hebrew, or Arabic, or
3. expand if it is predominately some Asian alphabet.

14.3 COMPRESSION OF UNICODE

A Unicode file is a sequence of characters in either UCS-2 or UTF-8 representation. On disk, these files are held as a sequence of bytes and can be compressed by any byte-oriented lossless compressor. Breaking the characters into their component bytes often degrades the compression from what might be expected. As this effect depends on the actual compressor, it is necessary to consider some of the main classes of compressor and how they might behave. The discussion assumes ASCII data, expanded to 16-bit UCS-2 symbols.

14.3.1 Finite-Context Statistical Compressors

These compressors, exemplified by traditional PPM compressors [2, 5], predict a symbol from a context of say the previous 4 symbols. That means 4 *bytes*, which in UCS-2 is only 2 *characters*. The compressor is then working at only half the expected order and may be expected to achieve rather poorer compression. While it is often possible to increase the order, that all too easily leads to a combinational explosion. While order 4 with 8-bit ASCII has a total of 4.3×10^9 possible contexts, order 4 with 16-bit Unicode has 1.8×10^{19} possible contexts. While no compressor will actually use all of these possible contexts, it must provide data structures to allow them, at least in some dense groups of contexts. There is no guarantee that a given implementation will permit this. With UTF-8 the context is difficult to determine. With English files it is probably unchanged, with European files it may be halved, while with Asian languages it may be reduced to one-third of that expected. (But with ideograms being equivalent to several letters, the compressor works at a much higher effective order than it would with letters and the degradation might be rather less.)

14.3.2 Unbounded-Context Statistical Compressors

These compressors, especially PPM* [6] and Burrows–Wheeler block sorting [4, 7], resemble their finite-context brethren, but have data structures or other techniques which allow contexts to grow indefinitely large. These compressors may be expected to adjust to the longer byte-wise contexts of Unicode by increasing the context order as needed. The operation may be slowed, but compression should remain good. Similar remarks apply to UTF-8 files.

14.3.3 LZ-77 Compressors

These have a sliding window or its equivalent of recent text and emit pointers into the window giving {phrase_displacement, phrase_length} couples. With a byte-oriented compressor, the UCS-2 symbols force both the displacement and the length to cover only half the expected number of symbols. Both of these effects reduce the compression, but only slightly. Byte-oriented UTF-8 may or may not have reduced coverage; the compressor can always detect character boundaries (and strictly speaking comparisons should always recognize the UTF-8 symbol alignment).

14.4 TEST COMPRESSORS

These tests use a variety of compressors and compression methods. While not all are state of the art, all but one are widely available and provide a good coverage of compression techniques. All are used with default options and parameters.

GZIP: The Unix compressor, released by the Free Software Foundation, is used as a good example of an LZ-77 class compressor.

BZIP: Seward's implementation of the Burrows–Wheeler block sorting compressor, also released by the Free Software Foundation [8]. (Note that this work used the original BZIP, rather than the later BZIP2.)

COMP-2: A PPM compressor described by Mark Nelson [3].

Compress: The well-known and traditional Unix compressor, implementing the LZW derivative of an LZ-78 compressor.

LZU: This is a special LZ-77 compressor, designed to operate in both 8-bit (LZU-8, for ASCII or UTF-8) and 16-bit (LZU-16, for UCS-2) modes and intended to compare similar compressors for the two modes. Although really designed only for UCS-2 compression, the 16-bit mode is tried for all files. The compressor is not optimized for good compression nor does it recognize byte order marks.

14.4.1 The Unicode File Test Suite

The Unicode file test suite is a multilingual collection of sample Unicode files gathered from various Unicode sources. The total length is about 85,000 characters, of which about 53,500 or 63% are ASCII. Other files have been simulated by expanding three text files from the Calgary Corpus, to provide a control of known text files.

The files are handled by a variety of techniques:

- All of the files are derived from UTF-8 (or ASCII) originals, expanded to UCS-2.
- The UTF-8 original files are compressed as bytes.
- The files, both UCS-2 and UTF-8, are compressed as bytes, with no regard to the 16-bit structure.
- The UCS-2 files are split into two component files, one containing the more significant bytes of each UCS-2 character and one containing the less significant bytes. The two components are individually compressed with a byte compressor and the total size of the two compressed files is combined. (For ASCII files one component is identical to the original and the other is all-zero.)

The important point is how a given compressor behaves on different versions of the same file rather than the relative performance of the compressors on each file—the differences *between* compressors are well known.

Data compression results are traditionally given in output bits per input byte, or bits per byte, with a recent move to give performance in bits per character (bpc). As Unicode no longer retains the identity between bytes and characters, the procedure in this paper is to present all results in bits per character, related to the 16-bit units of the UCS-2 files. Thus ASCII files are shown as bits per byte, while Unicode results are always in bits per Unicode character. The compression of UTF-8 files, considered as arbitrary files without respect to their encoded information, is a quite different matter, which is left until Section 14.6.

14.5 COMPARISONS

Results for the different compressors and file formats are shown in Table 14.4. In all cases the results are simply given as bits per character, averaged over each corpus. (The compression for UCS-32 often looks poor, but remember that it is given in bits per *two* bytes and the value must be halved for comparison with the traditional “bits per byte”.)

- The “split” files are not successful, because the division destroys too much contextual information.
- The constant-order PPM compressor shows the expected degradation on the UCS-2 files.
- Unix Compress also gives poor performance on the UCS-2 files.
- The two 8-bit LZ-77 compressors, GZIP and LZW-8, both give a small deterioration in UCS-2 compared with UTF-8.
- The 16-bit LZ-77 compressor LZW-16 gives quite different results for the different “endian” files, emphasizing the need to get the alignment correct, even in the full 16-bit character mode.

Table 14.4 Compression Results, in Bits per Character

		UTF-8	UCS-2 Big-endian	UCS-2 Little-endian	Split
COMP2 order-4	ASCII	2.38	2.95	2.95	2.38
Constant order PPM	Unicode	5.47	5.96	6.06	6.12
BZIP	ASCII	2.30	2.31	2.31	2.31
Burrows–Wheeler	Unicode	5.03	5.24	5.39	5.79
Compress	ASCII	3.66	4.55	4.55	3.72
LZ-78 or LZW	Unicode	7.87	7.91	7.30	7.30
GZIP	ASCII	2.67	3.25	3.25	2.68
LZ-77	Unicode	5.69	6.29	6.32	6.13
LZW-8	ASCII	3.24	4.30	4.30	3.24
LZ-77, 8-bit mode	Unicode	6.76	7.61	7.72	7.39
LZW-16	ASCII	4.21	3.39	3.65	4.20
LZ-77, 16-bit mode	Unicode	8.60	6.54	7.97	9.04

Table 14.5 Compression of UTF-8 Files (in Bits per Byte)

	BZIP	GZIP	Compress
Unicode	3.60	4.04	5.00
ASCII	2.30	2.67	3.66
Ratio Unicode: ASCII	1.56	1.52	1.37

A pleasing result is the performance of LZU-16 on UCS-2 files, compared with the similar LZU-8 on UTF-8 data. It shows that there may be considerable benefit from compressors which acknowledge the 16-bit structure of UCS-2 data.

14.6 UTF-8 COMPRESSION

As it is often necessary to compress information which is already in UTF-8 format, the compressibility of UTF-8 files is given for the three standard compressors, BZIP, GZIP, and Compress. The results, in Table 14.5, are now shown in bits per byte, this being a more appropriate measure for UTF-8 files of unknown content. (Remember that there is no simple relation between bytes and characters in UTF-8.)

Most of the files are reasonably compressible, though not to the extent usually expected for text files. The final file size is about 50% greater than would be expected for an ASCII text file (4.0 bpb vs 2.7 bpb for GZIP, or 3.6 vs 2.3 for BZIP). Although not apparent from Table 14.5, files which are essentially monoalphabetic in a Southeast Asian language may compress very well. Each character has a constant 2-byte prefix, which largely disappears with many compressors. A reasonable compression of say 3 bits per character then converts to a very respectable 1 bit per byte.

14.7 CONCLUSIONS

Unicode files have different compression characteristics from files of more traditional character representations. Accepted “good” compressors such as finite-context PPM do not necessarily work well, although unbounded context statistical compressors are quite satisfactory. Good dictionary or Liv–Zempel compressors also maintain their performance. Tests with a special test compressor indicate that better results may be obtained from compressors which work in the 16-bit units of canonical Unicode. Several possibilities arise for changing compressors to work more efficiently with Unicode files.

1. If a byte compressor detects that it is compressing a UCS-2 file, it could preprocess it into UTF-8 format.
2. Compressors could work with 16-bit symbols, much as LZU-16 has demonstrated. GZIP should certainly be a good candidate for extension, retaining its efficient output coding. The Burrows–Wheeler compressor (BZIP) may be amenable to 16-bit conversion, but at the cost of a slower and more difficult sort phase. The compressors may have to recognize the endian nature of files, to suit their coding details or internal data structures. A 16-bit compressor might well convert a UTF-8 (or ASCII) file to UCS-2 for compression.

Either of these conversions should yield compressors which are well tuned to the special requirements of Unicode data.

14.8 REFERENCES

1. The Unicode Standard, Version 2.0, 1996. *The Unicode Consortium*, Addison-Wesley, Reading, MA.
2. Bell, T. C., J. G. Cleary, and I. H. Witten, 1990. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ.
3. Nelson, M., 1991. Arithmetic coding and statistical modelling. *Dr. Dobbs Journal*, Feb 1991. Anonymous FTP from [wuarchive.wustl.edu/systems/msdos/msdos/ddjmag/ddj9102.zip](ftp://wuarchive.wustl.edu/systems/msdos/msdos/ddjmag/ddj9102.zip)
4. Burrows M., and D. J. Wheeler, 1994. A Block-sorting Lossless Data Compression Algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA. Available at gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z.
5. Cleary, J. G., and I. H. Witten, 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions and Communications*, COM-32, Vol. 4, pp. 396–402.
6. Cleary, J. G., W. T. Teahan, and I. H. Witten, Unbounded length contexts for PPM. *Data Compression Conference, DCC-95*, pp. 52–61.
7. Fenwick, P. M., 1996. The Burrows–Wheeler transform for block sorting text compression—Principles and improvements. *The Computer Journal*, Vol. 39, No. 9, pp. 731–740.
8. Seward, J., 1996. Private communication. For notes on the released BZIP see <http://hpux.cae.wisc.edu/man/Sysadmin/bzip-0.21>.

This Page Intentionally Left Blank

PART IV

Standards

This Page Intentionally Left Blank

JPEG-LS Lossless and Near Lossless Image Compression

MICHAEL W. HOFFMAN

15.1 LOSSLESS IMAGE COMPRESSION AND JPEG-LS

The original JPEG lossless compression component contained eight options for differentially encoding images. Seven predictors used some subset of neighboring pixels to form the prediction and the eighth used a zero value (no prediction). If an application permitted, the best of these predictors could be used for encoding an image.

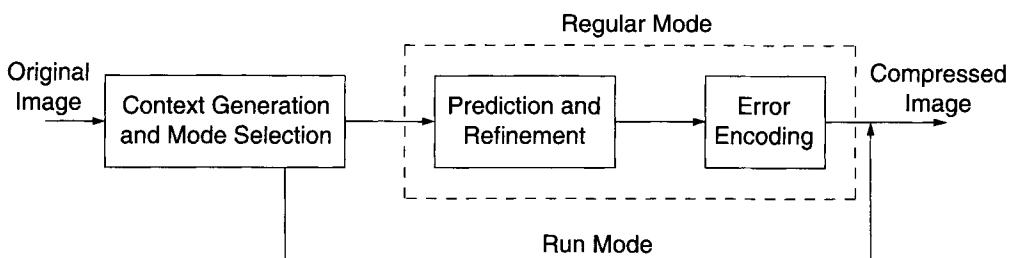
The Context Adaptive Lossless Image Compression (CALIC) algorithm was developed in response to a call for proposals for a replacement lossless image compression standard in 1994 [1, 2]. In the initial evaluation of competing proposals, CALIC came in first in six of seven categories. The JPEG-LS standard is derived from a Hewlett-Packard scheme called LOCO-I [3] that is much simpler than CALIC but which incorporates some of the key features of CALIC: context-based adaptive prediction with context-dependent adaptive bias removal.

In the following section the JPEG-LS algorithm is described. This includes an overview and descriptions of the context-based modeling, extension to multicomponent images, error correction and quantization, and entropy coding. The final section presents some example results using the JPEG-LS coding algorithm in both lossless and near lossless modes.

15.2 JPEG-LS

15.2.1 Overview of JPEG-LS

Figure 15.1 presents the overall block diagram of the JPEG-LS encoding algorithm. The main functional blocks consist of context modeling, mode determination (run versus regular), context-based prediction with a context-dependent error correction term, error quantization, and entropy coding. Since JPEG-LS has both lossless and near lossless modes, it is convenient to introduce a

**FIGURE 15.1**

Overall block diagram of JPEG-LS encoding.

term called NEAR that specifies the maximum absolute pixel error tolerated in the near lossless mode. Note that if NEAR is zero, then lossless encoding is achieved. In the context modeling a set of gradients is determined from reconstructed pixels adjacent to the pixel being encoded. If all of these local differences are less than NEAR, then near lossless coding is possible using the “run mode.” In this mode, runs of pixels that are close enough to their neighbors (i.e., within NEAR) are simply encoded as runs rather than as individual pixels. If this is not possible, then the pixels must be encoded individually in the “regular mode.”

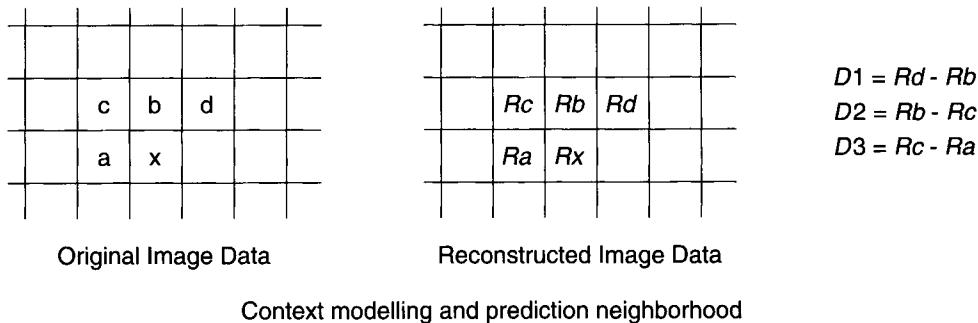
In the regular mode a context is obtained by classifying the local gradients and merging these classified groups into a reasonable number of contexts. An initial prediction is made based on an edge-detecting predictor that uses the values of the neighboring reconstructed pixels. This prediction is subsequently refined to remove the error bias associated with a given context. Then this error is mapped and quantized, followed by entropy coding.

In the case of multicomponent images, several options are available for encoding the image. First, each component can be encoded completely and separately, with the data for the second following the data for the first, etc. This is referred to as a single-component scan. Second, one of two multicomponent scans can be used, line interleaving or sample interleaving. There are slight differences in the determination of run and regular mode processing for the multicomponent scans (or modes). In the line-interleaved mode a predetermined number of lines from each component are encoded in a fashion very similar to the single-component mode. The data for the given number of lines in each component is encoded with the data for component 2 following that for component 1, and the data for component 3 following that for component 2, etc. In the sample-interleaved mode, pixels from each component are encoded sequentially, from one component to the next before the next pixel in the first component is encoded. This implies that for sample interleaving all of the components in a scan must have the same dimensions. The main difference in coding with sample interleaving is that runs must exist across all components before run mode processing is entered.

The JPEG-LS data stream includes markers, marker segments, and coded data segments. The special character $0xFF$ (i.e., FF in hexadecimal) is used to identify markers—with special restrictions on both coded image data and marker symbols that contain $0xFF$ to allow unambiguous decoding. Since the JPEG-LS encoder uses the reconstructed data available at the decoder, the decoding steps simply mirror those of the encoder to ensure lossless or near lossless reproductions of the original image data.

15.2.2 JPEG-LS Encoding

In this section the various blocks of the JPEG-LS encoding algorithm are discussed. Initially, the context determination and mode are described for single-component images. The extension of these procedures for multicomponent (e.g., color) images is described in the second part of this

**FIGURE 15.2**

Prediction neighborhood and gradient estimation based on reconstructed image data.

```

if (Di <= -T3) Qi = -4;
else if (Di <= -T2) Qi = -3;
else if (Di <= -T1) Qi = -2;
else if (Di <= -NEAR) Qi = -1;
else if (Di <= NEAR) Qi = 0;
else if (Di <= T1) Qi = 1;
else if (Di <= T2) Qi = 2;
else if (Di <= T3) Qi = 3;
else Qi = 4;

```

where $i=1,2,3$ for Di and Qi

$T1$, $T2$, and $T3$ are nonnegative thresholds

$NEAR$ is near lossless threshold (or 0 if lossless)

Context is $(Q1, Q2, Q3)$ {or $(-Q1, -Q2, -Q3)$ if first nonzero element is negative (with $SIGN = -1$)}
365 possible contexts

Gradient Quantization and Definitions of Context

FIGURE 15.3

Context definition from gradient data.

section. In the third part the prediction error correction, quantization, and reconstruction functions are described. In the final part of this section the entropy coding used in JPEG-LS is described.

15.2.2.1 *Context and Mode—Single-Component Images*

Figure 15.2 illustrates the prediction neighborhood and context basis of a pixel, x , to be encoded. Note that the encoder uses the reconstructed image data for context determination and prediction so that these processes can be duplicated at the decoder. Also shown in Fig. 15.2 are the formulas used to define the gradients (or pixel differences) $D1$, $D2$, and $D3$. For the first line in an image the reconstructed values Rb , Rc , and Rd are set to 0, as is the reconstructed value Ra for the first pixel. For the first (or last) pixel in subsequent lines, the reconstructed value Ra (or Rd) is set to the value of Rb , while Rc is set to the value assigned to Ra in the previous line.

The context is defined by quantizing the set of gradients. Given a set of gradients, Fig. 15.3 illustrates how the context ($Q1$, $Q2$, $Q3$) is derived. A set of known non-negative thresholds is defined as $T1$, $T2$, and $T3$. These thresholds are used to quantize each of the three gradients to one of nine levels. By using $NEAR$ as either 0 or the near lossless allowable distortion, this quantization of gradients can be applied to either lossless or near lossless modes of operations. Note that if the first non-zero context is negative, then the variable $SIGN$ is defined as -1 and the quantized gradients are stored as the negative of their actual values. Otherwise $SIGN$ is set to 1. This merging of contexts results in 365 rather than $729 (= 9^3)$ different contexts. Three hundred

sixty-four of these 365 possible contexts are stored in the variable arrays with indices ranging over $[0, \dots, 363]$. The all-zero context is used to detect transitions into run mode.

15.2.2.1.1 Run Mode If the three gradients are all equal to 0 (or less than NEAR for near lossless compression), then JPEG-LS enters a run mode in which runs of the same (or nearly the same) value as the previous reconstructed pixel are encoded. Otherwise, “regular mode processing” is used. In run mode processing, as long as the successive image pixels are equal to R_a (or within NEAR of R_a for the near lossless case), a run continues. The run is terminated by either the end of the current image line or a run interruption sample (i.e., one that violates the match condition to R_a defining the run). The coding for the run mode is broken into two sections: run coding and run interruption coding. Run-lengths in powers of 2 are encoded. In a “hit” situation, either the run-length is that power of 2 or an end of line has occurred. In a “miss” situation, the location of the termination of the run is transmitted. Run interruption samples can be one of two classes: those where the interruption sample has neighboring pixels R_a and R_b with values that are more than NEAR apart and those where this absolute difference is less than NEAR. Different predictors and a slightly different updating of context information are used in these cases—with the entropy coding being similar to that used in regular mode processing.

15.2.2.1.2 Regular Mode In regular mode processing, a median “edge-detecting” predictor is used. Figure 15.4 shows the pseudo-code used to determine the value of the prediction P_x for pixel x using the reconstructed values R_a , R_b , and R_c . This prediction is then corrected based on the context, as will be discussed shortly.

15.2.2.2 Modifications for Multicomponent Images

When more than a single image component exists, as is the case in color images, two multi-component scans can be used. The different components may have the same sampling resolution or they may have different resolutions. In the horizontal and vertical directions different sampling factors can be used. For an image with N_f components the sampling factors can be written as $H_0, H_1, \dots, H_{N_f-1}$ in the horizontal direction and $V_0, V_1, \dots, V_{N_f-1}$ in the vertical direction. Letting H_{max} and V_{max} be the maximum sampling factors in each direction, then the subsampling performed for the horizontal and vertical directions of component i relative to the highest resolution component can be denoted as H_i/H_{max} and V_i/V_{max} , respectively. Figure 15.5 illustrates an

```

if ( $R_c \geq \max(R_a, R_b)$ ) {
     $P_x = \min(R_a, R_b);$ 
} else {
    if ( $R_c \leq \min(R_a, R_b)$ ) {
         $P_x = \max(R_a, R_b);$ 
    } else {
         $P_x = R_a + R_b - R_c;$ 
    }
}

```

Predicted Value for Pixel x

FIGURE 15.4

Edge-detecting predictor used to generate the prediction P_x .

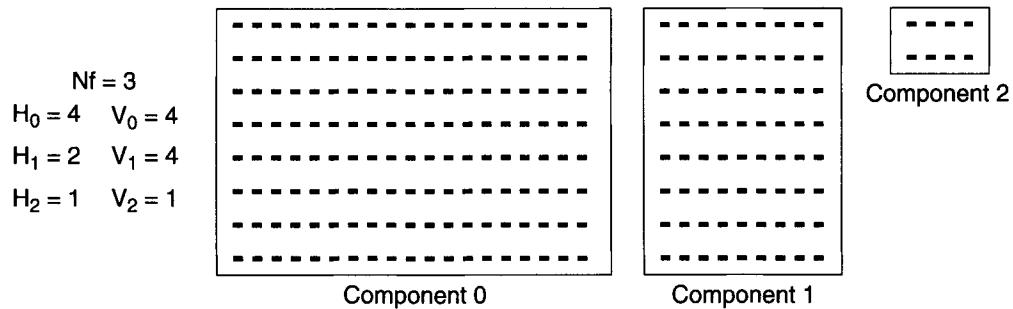


FIGURE 15.5

Example sampling for a three-component image.

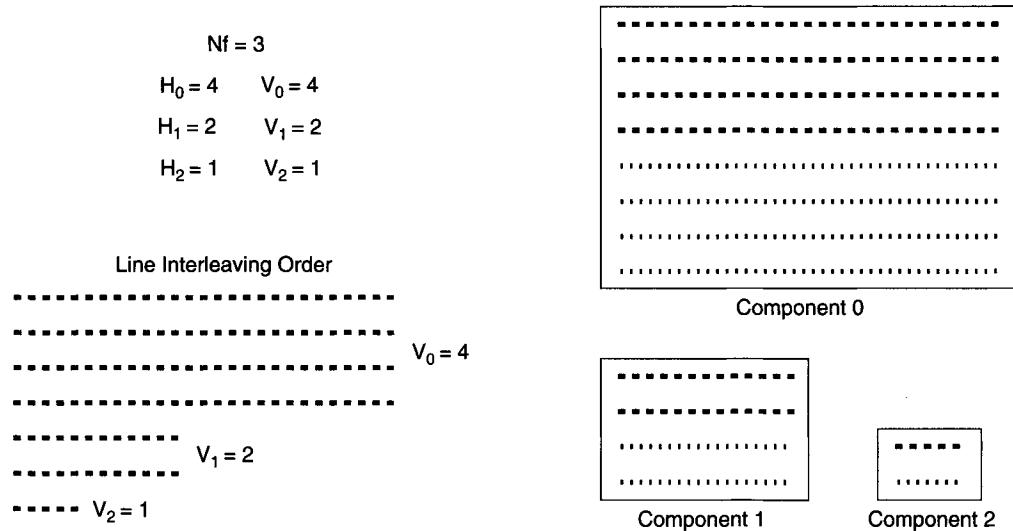


FIGURE 15.6

Example line interleaving for a three-component image.

example multicomponent image that contains three components. The larger the sampling factor, the more lines in that dimension for a particular component. For a red-green-blue color image, these sampling factors may be the same for each of the three components. In a Y-U-V color image, the luminance component, Y, may have higher sampling factors than the two chrominance components, U and V. (This representation exploits the relative spatial sensitivity to chrominance changes with respect to the luminance spatial sensitivity.) There are two ways a multicomponent image can be interleaved: line interleaving and sample interleaving.

15.2.2.2.1 Line Interleaving In line interleaving, the vertical sampling factors for each component determine the number of lines coded for that component before lines are interleaved from the next component. Figure 15.6 illustrates line interleaving for a given multicomponent image example. In the example $V_0 = 4$, four lines are coded from component 0. $V_1 = 2$, so two lines of component 1 are coded (note that these are shorter lines). Finally, $V_2 = 1$ and a single line is coded from component 2. In line interleaving the determination of run versus regular mode processing

is identical to the process used for single-component images. Recall that in single-component images the run versus regular mode processing decision was determined within lines of the image—so the extension to line interleaving is straightforward.

15.2.2.2.2 Sample Interleaving For a sample-interleaved multicomponent image, pixels are interleaved from one component to the next; i.e., one pixel is taken from each component sequentially. For this interleaving mode to make sense, the components must have identical sampling factors (as in red-green-blue color images). The run versus regular mode processing decision is slightly modified to accommodate sample interleaving. For a run to be declared it must exist across *all the image components*. When the run ends in any one of the components, i.e., the difference between the pixels being coded and R_a exceeds NEAR (or zero in lossless coding) in any component, then the run is broken and the encoder returns to regular mode processing. One side effect of this is that the all-zeros context must be included for the predictor since several image components may have a zero gradient but another component's value may preclude the use of run mode processing.

15.2.2.3 Prediction Error Correction, Quantization, and Reconstruction

The edge-detecting predictor used in regular mode processing produces an initial estimate of pixel value x , P_x . This estimate is corrected based on the context of the given pixel. A JPEG-LS encoder keeps track of several arrays of variables that are indexed by the context, Q . The arrays include $N[Q]$, $A[Q]$, $B[Q]$, and $C[Q]$. $N[Q]$ is increased by 1 each time the given context Q is used (this is initialized to the value 1). $A[Q]$ is used to track the absolute normalized (by quantizer step size) error to determine the parameters used in entropy coding. $B[Q]$ accumulates the non-normalized pixel error bias (positive or negative) and it is used to increment or decrement $C[Q]$, the error correction term for context Q . These values are updated after the error for a given pixel has been computed. When $N[Q]$ exceeds a RESET value, $A[Q]$, $B[Q]$, and $N[Q]$ are right-shifted by 1 bit to halve their contents and avoid numerical problems. Note that all of these context-indexed arrays use the quantized variables (such as the prediction error) that are available to both the encoder and the decoder.

When the value of $B[Q]$ falls outside of the range $[-N[Q] + 1, 0]$ its value is mapped back into this interval. If $B[Q]$ is less than or equal to $-N[Q]$, then $N[Q]$ is added to it and $C[Q]$ is decremented by 1; i.e., the error correction term is reduced. If $B[Q]$ is greater than 0, then $N[Q]$ is subtracted from it and $C[Q]$ is incremented by 1. After these operations $B[Q]$ is hard-limited to the nearest end-point of the $[-N[Q] + 1, 0]$ interval. In this way, the bias term continues to push the error correction term up or down depending on how consistently and significantly the error tends to be positive or negative for a short run of data. The maximum change in the value of $C[Q]$ is ± 1 for any one sample error estimate. Figure 15.7 gives the correction to a pixel prediction, P_x , for a given context, Q , based on the error bias correction contained in an array $C[Q]$.

The error is either directly coded for lossless compression or quantized to a step size of $(2\text{NEAR} + 1)$ for near lossless compression. The representation inside the encoder is normalized to an integer label in which the value 1 corresponds to the step size (this is 1 for lossless compression). The encoder requires the reconstructed value that will be produced at the decoder. This is obtained by multiplying the normalized error value by the error SIGN and the step size of the quantizer and adding this to the predicted value (including corrections) produced by both the encoder and the decoder. At this point the error value is mapped to a non-negative number to facilitate entropy coding.

```

if (SIGN == +1) {
    Px = Px + C[Q];
} else {
    Px = Px - C[Q];
}
if (Px > MAX) {
    Px = MAX;
} else if (Px < 0) {
    Px = 0;
}

```

Correction for Pixel x Prediction

FIGURE 15.7

Prediction error correction based on context Q accumulated error bias correction, $C[Q]$. MAX is the maximum pixel value.

15.2.2.4 JPEG-LS Entropy Coding

The entropy coding in JPEG-LS can be conveniently separated into regular mode coding of mapped error values and run mode coding of run-lengths and run interruption samples. In this section regular mode coding is discussed first, followed by a discussion of run mode coding.

15.2.2.4.1 Regular Mode Coding Limited-length Golomb codes are used to represent the mapped (non-negative) error values in regular mode processing. The codewords can be thought of as two separate codes: one for the remainder of the error value divided by 2^k and one for the result of the integer division of the error value by 2^k . k is a code parameter that specifies the number of bits used to represent the remainder. The code for the k remainder bits is simply a fixed-length k -bit unsigned binary representation. The code for the integer division result is a unary code that terminates in a 1. For example, to represent 37 with a $k = 3$ code the unary part is given by $37/8 = 4$ zeros and a one (00001) while the remainder is the 3-bit representation of 5 (101). The entire codeword is obtained as 00001101.

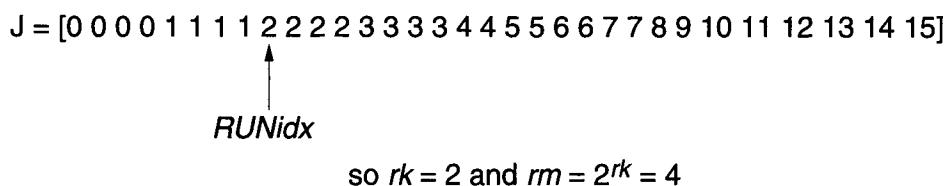
Figure 15.8 illustrates the codewords for a $k = 2$ with a maximum length limit of 32 bits. The unary code represents the result of integer division of the mapped errors by 4 ($= 2^2$). To resolve the ambiguity associated with the many-to-one mapping provided by the integer division, the 2-bit unsigned binary code represents the remainder. The use of the unary code results in very large codewords for large error values. The JPEG-LS entropy coding imposes an upper limit on the number of bits that can be used for a codeword. This is done by using an escape code after which the L -bit value of the mapped error is directly represented by an unsigned binary code. L is determined by the number of bits required to represent the quantizer error values. In the example the maximum number of bits in a codeword is 32 and the value of L is 8. An example of a mapped error of 123 is shown. For this mapped error value an escape code of 23 zeros followed by a one indicates the exception and this is followed by the unsigned 8-bit binary representation of $123 - 1$. Since there is never a reason to code a mapped error value of 0 using an escape code, the mapped errors are coded as their value less 1.

15.2.2.4.2 Run Mode Coding When run mode processing is entered, a different form of coding is used. An internal array of values defines a variety of length of runs. Figure 15.9 illustrates an array J that contains 32 values of rk . Potential runs are compared to the current run-length rm .

Prediction Error	Mapped Error	Codewords (2 parts)	
		Unary	Remainder
0	0	1	00
-1	1	1	01
1	2	1	10
-2	3	1	11
2	4	01	00
-3	5	01	01
3	6	01	10
-4	7	01	11
4	8	001	00
-5	9	001	01
5	10	001	10
-6	11	001	11
6	12	0001	00
-7	13	0001	01
7	14	0001	10
-8	15	0001	11
8	16	00001	00
-9	17	00001	01
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
Example	-62	123	0000000000000000000000000001 (escape) 01111010 (8 bit representation of 123-1) total of 32 bits

FIGURE 15.8

Example error values, mapped error values, and codewords for JPEG-LS error coding.

*RUNidx* points to the value of *rk* which sets the value of *rm***FIGURE 15.9**

Run mode coding length determination.

rm is defined as 2 to the power *rk*, which is the array value currently pointed to by the pointer *RUNidx*. In Fig. 15.9, *RUNidx* is 8, which results in an *rk* value of 2 and an *rm* value of 4. If run mode processing is entered, the current value of *rm* is compared to the length of the current run. If the run is equal to or in excess of *rm*, then a hit is declared and a 1 is transmitted to

indicate this hit. At this point the value of $RUNidx$ is incremented by 1 and the process continues. If an end of line is reached during a run that has not reached rm in length, a 1 is also transmitted; however, the variable $RUNidx$ is not changed. If the run ends in a length less than rm , then a miss is indicated by the transmission of a 0 bit. After this bit rk bits are transmitted to indicate the length of the run (this must be $\leq 2^{rk} - 1$). The value of $RUNidx$ after a miss is decreased by 1. The coding of the run interruption sample is similar to regular mode coding except that two separate contexts are reserved for determining the coding used on this sample. In addition, the prediction uses either the pixel Ra that started the run or its neighbor Rb as the prediction without further correction.

15.2.3 JPEG-LS Decoding

Since the JPEG-LS encoding algorithm uses information available to the decoder, the decoding process recapitulates many of the basic encoder functions. After decoding the entropy data, the decoder either completes the run replication (in run mode) or reconstructs the error values (in regular mode) and adds these to the predicted values (with the same corrections provided at the encoder) to generate the image. Note that the context definitions and updates in the decoder are also identical to those produced in the encoder.

15.3 SUMMARY

Figure 15.10 illustrates near lossless coding for the *sensin* test image (8-bit grayscale). The original image is shown on the top left (NEAR = 0). The reconstructed images for NEAR = 3, 7, and 15 are shown on the top right, bottom left, and bottom right, respectively. Note the degradation in image quality—particularly in flat regions such as the blackboard—as the value of NEAR becomes larger.

The compression attained on the *sensin* image for varying values of NEAR is illustrated in Table 15.1. By removing the restriction of lossless compression and replacing it with a near lossless constraint that the maximum difference allowed in defining a run is 1, the compressed file size drops nearly 40%, a savings of about 1.4 bits per pixel. By increasing NEAR to 3, an additional 0.75 bits per pixel of compression are obtained. As was seen in Fig. 15.10, for larger values of NEAR the additional compression probably is not worth the artifacts introduced by the lossy coding—i.e., it is not near lossless compression, rather it is lossy compression. At these lower rates, better image compression strategies exist [4].

Table 15.1 Compression in Bits per Pixel and Percentage of Original as NEAR Is Varied from 0 (Lossless) to 15 for the 8-Bit Grayscale Image *sensin*

NEAR	Bits/Pixel	% of Original
0	3.704	46.3%
1	2.281	28.5%
3	1.528	19.1%
7	1.093	13.7%
15	0.741	9.3%



FIGURE 15.10

The *sensin* test image (top left), NEAR = 3 (top right), NEAR = 7 (bottom left), NEAR = 15 (bottom right).

15.4 REFERENCES

1. Wu, X., N. D. Memon, and K. Sayood, 1995. A Context Based Adaptive Lossless/Nearly Lossless Coding Scheme for Continuous Tone Images, ISO Working Document, ISO/IEC SC29/WG1/N256.
2. Wu, X., and N. D. Memon, 1996. CALIC—A context based adaptive lossless coding scheme. *IEEE Transactions on Communications*, vol. 45, pp. 437–444, May 1996.
3. Weinberger, M., G. Seroussi, and G. Sapiro, 1996. LOCO-I: A low complexity, context-based, lossless image compression algorithm, In *Proceedings of Data Compression Conference*, Snowbird, UT, March–April 1996.
4. Sayood, K., 2000. *Introduction to Data Compression*, 2nd ed., Morgan-Kaufmann, San Mateo, CA.
5. ITU-T Recommendation T.87. 2000. Information Technology—Lossless and near lossless compression of continuous tone still images, *JPEG-LS Recommendation*.

The CCSDS Lossless Data Compression Recommendation for Space Applications¹

PEN-SHU YEH

16.1 INTRODUCTION

In the late 1980s, when the author started working at the Goddard Space Flight Center for the National Aeronautics and Space Administration (NASA), several scientists were in the process of formulating the next generation of Earth-viewing science instruments, such as the Moderate Resolution Imaging Spectroradiometer. The instrument would have over 30 spectral bands and would transmit an enormous amount of data through the communications channel. In an attempt to solve the problem associated with buffering a large amount of data on board and transmitting it to ground stations, the author was assigned the task of investigating lossless compression algorithms for space implementation to compress science data in order to reduce the requirement on bandwidth and storage.

Space-based algorithm implementations face extra constraints relative to ground-based implementations. The difference is mainly to the lack of computing resources in space and possible bit errors incurred in the communication channel. A spacecraft is a physically compact environment where each subsystem (power, attitude control, instrument, data handling, telemetry, etc.) is given only limited budget in terms of power, mass, and weight. Any computation other than that which is absolutely necessary is normally not supported. Taking all these constraints into consideration, a set of requirements was first formulated for selecting a lossless compression algorithm:

- a. The algorithm must adapt to changes in data statistics in order to maximize compression performance.
- b. The algorithm must be implemented in real time with small memory and little power usage.

¹ Author's note: Part of the material from this chapter is taken from the Consultative Committee for Space Data Systems (CCSDS) publications [1, 2] for which the author is a major contributor.

- c. The algorithm must interface with a packet data system such that each packet can be independently decoded without requiring information from other packets.

At the time of the study (early 1990s), the “real time” requirement in (b) required over 7.5 million samples/s for a 30 frames/s 512^2 charge-coupled device sensor. Requirement (c) allows a long scan-line of pixels to be compressed independently into a data packet, which will then be protected by an error control coding scheme. This requirement constrains error propagation within a packet when an uncorrectable channel error occurs.

Several available algorithms were evaluated on test science data: Huffman [3], Ziv–Lempel (see Ref. [4]), arithmetic [5], and Rice [6]. The excellent compression performance and the high throughput rate of the Rice algorithm suggested further study, which resulted in a mathematical proof [7] of its performance. The proof establishes the Rice algorithm as a type of adaptive Huffman code on decorrelated data with a Laplacian distribution. It so happens that most of the data collected on science instruments, be it one- or two-dimensional, or even multispectral, once decorrelated, fits a Laplacian distribution.

Further study on the algorithm brought out a parallel architecture which was then implemented in an application-specific integrated circuit for space applications [8]. An extension to low-entropy data was also devised and incorporated in the original Rice architecture. The resulting algorithm is referred to as the extended_Rice, or e_Rice, algorithm.

In 1994, with sufficient interest from the international space agencies, the e_Rice algorithm was proposed to the CCSDS subpanel as a candidate for recommendation as a standard. In 1997, CCSDS published the Blue Book [1], which adopted the e_Rice algorithm as the recommendation for the standard. A Green Book [2], which serves as an application guide, was also released.

16.2 THE e_RICE ALGORITHM

The e_Rice algorithm exploits a set of variable-length codes to achieve compression. Each code is nearly optimal for a particular geometrically distributed source. Variable-length codes, such as Huffman codes and the codes used by the Rice algorithm, compress data by assigning shorter codewords to symbols that are expected to occur with higher frequency, as illustrated in Section 16.3.1. By using several different codes and transmitting the code identifier, the e_Rice algorithm can adapt to many sources from low entropy (more compressible) to high entropy (less compressible). Because blocks of source samples are encoded independently, side information does not need to be carried across packet boundaries; if decorrelation of source samples is executed only among these blocks, then the performance of the algorithm is independent of packet size.

A block diagram of the e_Rice algorithm is shown in Fig. 16.1. It consists of a preprocessor to decorrelate data samples and subsequently map them into symbols suitable for the entropy coding stage. The input data to the preprocessor, x , is a J -sample block of n -bit samples:

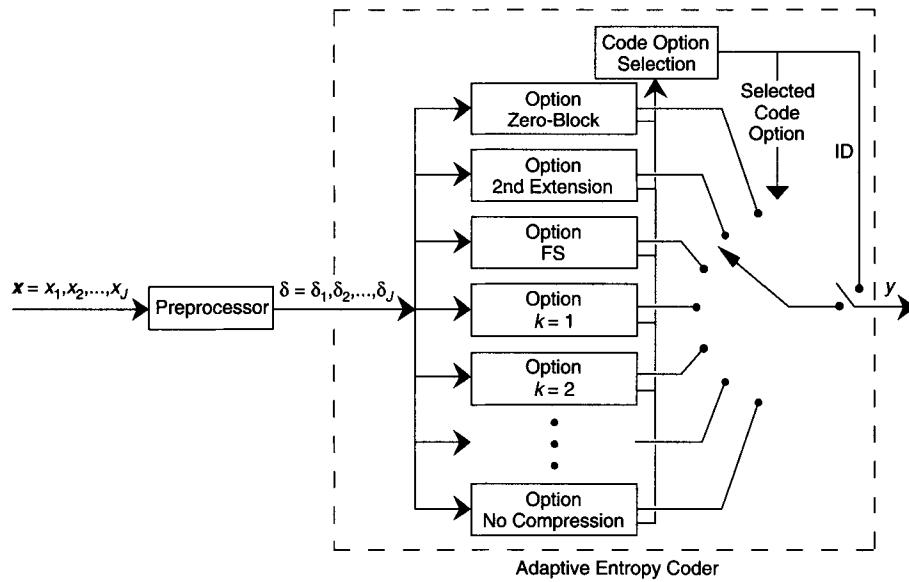
$$x = x_1, x_2, \dots, x_J.$$

The preprocessor transforms input data into blocks of preprocessed samples, δ , where

$$\delta = \delta_1, \delta_2, \dots, \delta_J.$$

The Adaptive Encoder converts preprocessed samples, δ , into an encoded bit sequence y .

The entropy coding module is a collection of variable-length codes operating in parallel on blocks of J preprocessed samples. The coding option achieving the highest compression is

**FIGURE 16.1**

The encoder architecture.

selected for transmission, along with an ID bit pattern used to identify the option to the decoder. Because a new compression option can be selected for each block, the e_Rice algorithm can adapt to changing source statistics. Although the CCSDS Recommendation specifies that the parameter J be either 8 or 16 samples per block, the preferred value is 16. The value of 16 samples per block is the result of experiments performed on several classes of science data, both imaging and non-imaging. These studies monitored the achievable compression ratio as a function of the parameter J , which was set to 8, 16, 32, and 64 samples/block for the various classes of science data. Values of J less than 16 result in a higher percentage of overhead, which yields a lower compression ratio, whereas values of J higher than 16 yield low overhead but have less adaptability to variations in source data statistics.

16.3 THE ADAPTIVE ENTROPY CODER

16.3.1 Fundamental Sequence Encoding

To see how a variable-length code can achieve data compression, consider two methods of encoding sequences from an alphabet of four symbols:

Symbol	Probability	Code 1	Code 2
s_1	0.6	00	1
s_2	0.2	01	01
s_3	0.1	10	001
s_4	0.1	11	0001

Code 2 is known as the “comma” code; it is also defined as the Fundamental Sequence (FS) code.

Table 16.1 Fundamental Sequence Codewords as a Function of the Preprocessed Samples

Preprocessed Sample Values, δ_i	FS Codeword
0	1
1	01
2	001
:	:
$2^n - 1$	<u>0000...00001</u> ($2^n - 1$ zeros)

Table 16.2 Examples of Split-Sample Options Using Fundamental Sequence Codes

Sample Values	4-bit Binary Representation	FS Code, $k = 0$	$k = 1$ 1 LSB + FS Code	$k = 2$ 2 LSB + FS Code
8	1000	00000001	0 00001	00 001
7	0111	00000001	1 0001	11 01
1	0001	01	1 1	01 1
4	0100	00001	0 001	00 01
2	0010	001	0 01	10 1
5	0101	000001	1 001	01 01
0	0000	1	0 1	00 1
3	0011	0001	1 01	11 1
Total bits	32	38	29	29

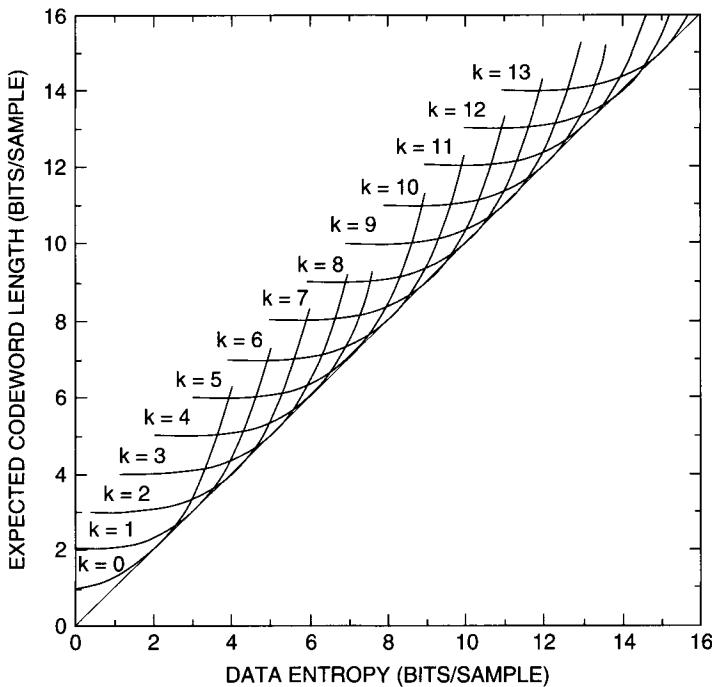
Using either code, encoded bitstreams corresponding to symbol sequences are uniquely decodable. For Code 1, we recognize that every two encoded bits correspond to a symbol. For the FS code we recognize that each “1” digit signals the end of a codeword, and the number of preceding zeros identifies which symbol was transmitted. This simple procedure allows FS codewords to be decoded without the use of lookup tables.

The reason that the FS code can achieve compression is that when symbol s_1 occurs very frequently, and symbols s_3 and s_4 are very rare, on average we will transmit fewer than two encoded bits per symbol. In the above example, Code 2 will achieve an average codeword length of $(0.6 + 2 \times 0.2 + 3 \times 0.1 + 4 \times 0.1) = 1.7$ bits per symbol, whereas Code 1 always requires two encoded bits per symbol.

Longer FS codes achieve compression in a similar manner. Table 16.1 illustrates the FS codewords for preprocessed sample values with an n -bit dynamic range.

16.3.2 The Split-Sample Option

Most of the options in the entropy coder are called “split-sample options.” The k th split-sample option takes a block of J preprocessed data samples, splits off the k least significant bits (LSBs) from each sample, and encodes the remaining higher order bits with a simple FS codeword before appending the split bits to the encoded FS datastream. This is illustrated in Table 16.2 for the case of k split bit being 0 (no split-bit), 1, or 2 on a sequence of 4-bit samples.

**FIGURE 16.2**

Performance curve for k .

From Table 16.2 either $k = 1$ or $k = 2$ will achieve data reduction from the original 32 bits to 29 bits. As a convention, when a tie exists, the option with smaller k value is chosen. In this case, $k = 1$ will be selected. When a block of J samples is coded with one split-sample option, the k split bits from each sample are concatenated using the data format specified in Section 16.5.

Each split-sample option in the Rice algorithm is designed to produce compressed data with an increment in the codeword length of about 1 bit/sample (approximately $k + 1.5$ to $k + 2.5$ bits/sample); the code option yielding the fewest encoded bits will be chosen for each block by the option-select logic. This option selection process ensures that the block will be coded with the best available code option on the same block of data, but this does not necessarily indicate that the source entropy lies in that range. The actual source entropy value could be lower; the source statistics and the effectiveness of the preprocessing stage determine how closely entropy can be approached.

A theoretical expected codeword length of the split-sample coding scheme is shown in Fig. 16.2 for various values of k , where $k = 0$ is the fundamental sequence option. These curves are obtained under the assumption that the source approaches a discrete geometric distribution. With this source model, tracing only the portions of these curves that are closest to the ideal diagonal performance curve at any given entropy will result in a theoretically projected performance curve as shown in Fig. 16.3. For a practical system as shown in Fig. 16.1, ID bits of 3- or 4-bit length will be needed for every block of J samples for cases of an 8-option or a 16-option system. In such a system, the source is likely to deviate from the discrete geometric distribution; however, the option with the fewest coded bits will be selected. Actual coding results on aerial imagery, along with the coder's applicability to other data sources with a Gaussian or a Poisson probability distribution function, are provided in Ref. [9].

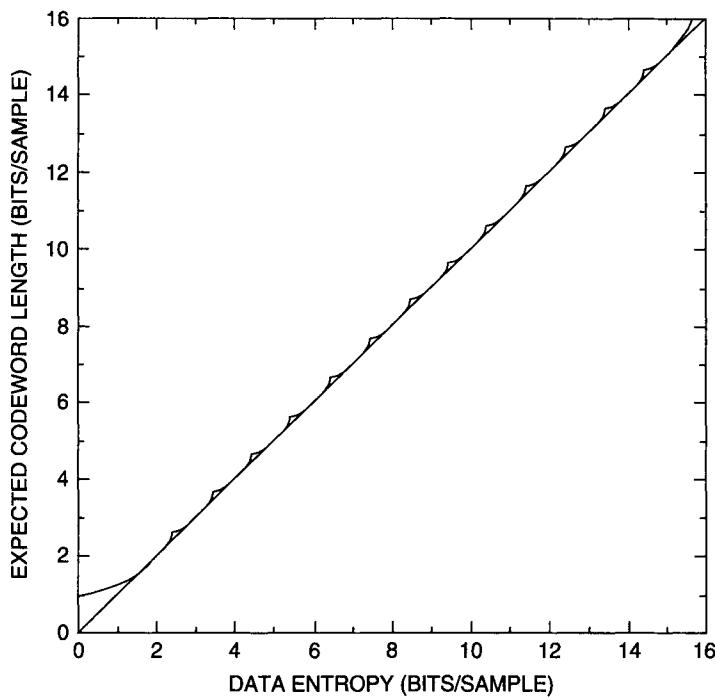


FIGURE 16.3
Effective performance curve.

16.3.3 Low-Entropy Options

16.3.3.1 General

The two code options, the Second-Extension option² and the Zero-Block option, provide more efficient coding than other options when the preprocessed data is highly compressible.

16.3.3.2 The Second-Extension Option

When the Second-Extension option is selected, each pair of preprocessed samples in a J -sample block is transformed and encoded using an FS codeword. Let δ_i and δ_{i+1} be adjacent pairs of samples from a J -sample preprocessed data block. They are transformed into a single new symbol, γ , by the following equation:

$$\gamma = (\delta_i + \delta_{i+1})(\delta_i + \delta_{i+1} + 1)/2 + \delta_{i+1}.$$

The $J/2$ transformed symbols in a block are encoded using the FS codeword of Table 16.1. The above process requires J to be an even integer which the recommended values obey ($J = 8$ or 16).

16.3.3.3 Zero-Block Option

The idea of the Zero-Block option was first suggested by J. Venbrux at the University of Idaho's Microelectronics Research Laboratory to deal with long runs of 0's. It is selected when one or

² The first extension of a preprocessed sample is the preprocessed sample itself.

Table 16.3 Zero-Block Fundamental Sequence Codewords as a Function of the Number of Consecutive All-Zeros Blocks

Number of All-Zeros Blocks	FS Codeword
1	1
2	01
3	001
4	0001
ROS	00001
5	000001
6	0000001
7	00000001
8	000000001
:	:
63	0000...000000001 (63 0's and a 1)

more blocks of preprocessed samples are all zeros. In this case, a single codeword may represent several blocks of preprocessed samples, unlike other options where an FS codeword represents only one or two preprocessed samples.

The set of blocks between consecutive reference samples, r , as described in Section 16.4.2, is partitioned into one or more segments. Each segment, except possibly the last, contains s blocks. The recommended value of s is 64.

Within each segment, each group of adjacent all-zeros blocks is encoded by the FS codewords, specified in Table 16.3, which identify the length of each group. The Remainder-of-Segment (ROS) codeword in Table 16.3 is used to indicate that the remainder of a segment consists of five or more all-zeros blocks.

16.3.4 No Compression

The last option is not to apply any data compression. If it is the selected option, the preprocessed block of samples receives an attached identification field but is otherwise unaltered.

16.3.5 Code Selection

The Adaptive Entropy Coder includes a code selection function, which selects the coding option that performs best on the current block of samples. The selection is made on the basis of the number of bits that the selected option will use to code the current block of samples. An ID bit sequence specifies which option was used to encode the accompanying set of codewords. The ID bit sequences are shown in Table 16.4 for a sample dynamic range up to 32 bits.

For applications not requiring the full entropy range of performance provided by the specified options, a subset of the options at the source may be implemented. The ID specified in Table 16.4 is always required and suggested, even if a subset of the options is used. With this ID set, the same standard decoder can be used for decoding partial implementations of the encoder.

Table 16.4 Selected Code Option Identification Key

Option	$n \leq 8$	$8 < n \leq 16$	$16 < n$
Zero-Block	0000	00000	000000
Second-Extension	0001	00001	000001
FS	001	0001	00001
$k = 1$	010	0010	00010
$k = 2$	011	0011	00011
$k = 3$	100	0100	00100
$k = 4$	101	0101	00101
$k = 5$	110	0110	00110
$k = 6$	—	0111	00111
$k = 7$	—	1000	01000
$k = 8$	—	1001	01001
$k = 9$	—	1010	01010
$k = 10$	—	1011	01011
$k = 11$	—	1100	01100
$k = 12$	—	1101	01101
$k = 13$	—	1110	01110
$k = 14$	—	—	01111
$k = 15$	—	—	10000
⋮	⋮	⋮	⋮
$k = 29$	—	—	11110
No compression	111	1111	11111

Note: A dash indicates no applicable value.

16.4 PREPROCESSOR

The role of the preprocessor is to transform the data into samples that can be more efficiently compressed by the entropy encoder. To ensure that compression is “lossless,” the preprocessor must be reversible. The FS and sample-split options work on non-negative integer values. The most effective preprocessor will transform integer data values into all non-negative values with a unimodal distribution resembling a one-sided Laplace function. The example in Section 16.3.1 shows that to achieve effective compression, we need a preprocessing stage that transforms the original data so that shorter codewords occur with higher probability than longer codewords. If there are several candidate preprocessing stages, we would like to select the one that produces the shortest average codeword length.

In general a preprocessor that removes correlation between samples in the input data block will improve the performance of the entropy coder. In the following discussion, we assume that preprocessing is done by a predictor followed by a prediction error mapper. For some types of data, more sophisticated transform-based techniques can offer improved compression efficiency, at the expense of higher complexity.

16.4.1 Predictor

The decorrelation function of the preprocessor can be implemented by a judicious choice of predictor for an expected set of data samples. The selection of a predictor should take into account the expected data as well as possible variations in the background noise and the gain of

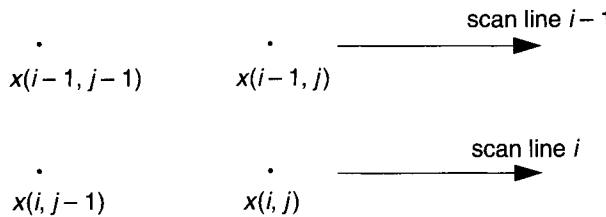


FIGURE 16.4
Typical pixel data arrangement.

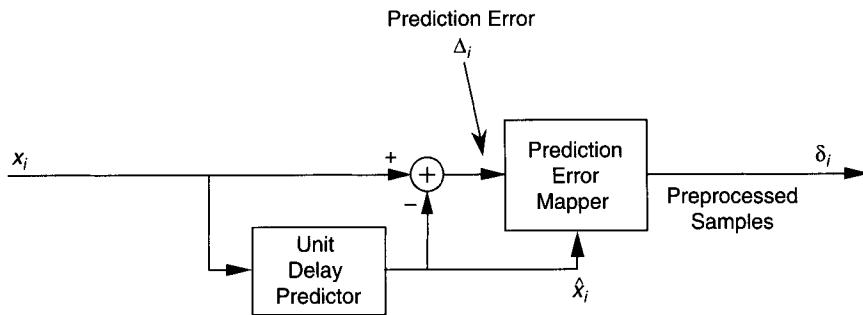


FIGURE 16.5
Preprocessor using a unit-delay predictor.

the sensors acquiring the data. The predictor should be chosen to minimize the amount of noise resulting from sensor non-uniformity.

Given the data arrangement in Fig. 16.4, the predictor value $\hat{x}(i, j)$ of input value $x(i, j)$ can be

- (a) A one-dimensional first-order predictor: $\hat{x}(i, j) = x(i, j - 1)$, or $\hat{x}(i, j) = x(i - 1, j)$.
- (b) A two-dimensional predictor: $\hat{x}(i, j) = 1/2[x(i, j - 1) + x(i - 1, j)]$.
- (c) A two-dimensional predictor: $\hat{x}(i, j) = 1/8[3x(i, j - 1) + 3x(i - 1, j) + 2x(i - 1, j - 1)]$.

Other types of weighted predictor can be devised depending on applications. For multispectral data, another prediction technique is to use samples from one spectral band as an input to a higher order prediction function for the next band.

One of the simplest predictive coders is a linear first-order unit-delay predictor shown in Fig. 16.5. The output, Δ_i , will be the difference between the input data sample and the preceding data sample.

16.4.2 Reference Sample

It is clear from Fig. 16.5 that for a reversible operation, a reference sample is needed to perform the first prediction. A reference sample is an unaltered data sample upon which successive predictions are based. Reference samples are required by the decoder to recover the original values from difference values. In cases where a reference sample is not used in the preprocessor or where

the preprocessor is absent, it shall not be inserted. The user must determine how often to insert references. When required, the reference must be the first sample of a block of J input data samples. In packetized formats, the reference sample must be in the first Coded Data Set (CDS) in the packet data field, as defined in Section 16.5, and must be repeated after every r blocks of data samples.

16.4.3 Prediction Error Mapper

Based on the predicted value, \hat{x}_i , the prediction error mapper converts each prediction error value, Δ_i , to an n -bit non-negative integer, δ_i , suitable for processing by the entropy coder. For most efficient compression by the entropy coding stage, the preprocessed symbols, δ_i , should satisfy

$$p_0 \geq p_1 \geq p_2 \geq \dots p_j \geq \dots p_{(2^n-1)},$$

where p_j is the probability that δ_i equals integer j . This ensures that more probable symbols are encoded with shorter codewords.

The following example illustrates the operation of the prediction error mapper after a unit-delay predictor is applied to 8-bit data samples of values from 0 to 255:

Sample Value x_i	Predictor Value \hat{x}_i	Δ_i	θ_i	δ_i
101	—	—	—	—
101	101	0	101	0
100	101	-1	101	1
101	100	1	100	2
99	101	-2	101	3
101	99	2	99	4
223	101	122	101	223
100	223	-123	32	155

If we let x_{\min} and x_{\max} denote the minimum and maximum values of any input sample x_i , then clearly any reasonable predicted value \hat{x}_i must lie in the range $[x_{\min}, x_{\max}]$. Consequently, the prediction error value Δ_i must be one of the 2^n values in the range $[x_{\min} - \hat{x}_i, x_{\max} - \hat{x}_i]$, as illustrated in Fig. 16.6. We expect that for a well-chosen predictor, small values of $|\Delta_i|$ are more likely than large values, as shown in Fig. 16.7. Consequently, the prediction error mapping function

$$\delta_i = \begin{cases} 2\Delta_i & 0 \leq \Delta_i \leq \theta \\ 2|\Delta_i|-1 & -\theta \leq \Delta_i < 0 \\ \theta + |\Delta_i| & \text{otherwise,} \end{cases}$$

where $\theta = \min(x_{\max} - \hat{x}_i, \hat{x}_i - x_{\min})$, has the property that $p_i < p_j$ whenever $|\Delta_i| > |\Delta_j|$. This property increases the likelihood that the probability ordering of p_i (described above) is satisfied.

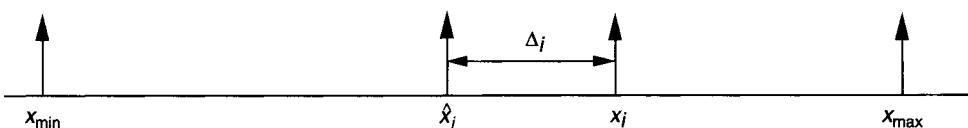
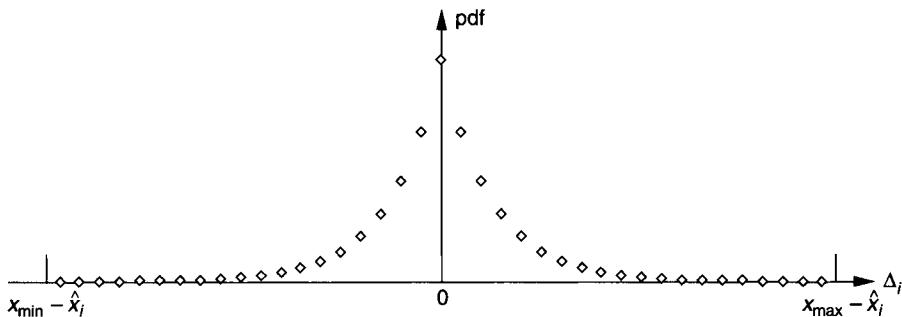


FIGURE 16.6

Relative position of sample value x_i and predictor value \hat{x}_i .

**FIGURE 16.7**

Typical probability distribution function (pdf) of Δ_i for imaging data.

16.5 CODED DATA FORMAT

The coded bits from each block of J samples follow a predefined format so that cross-support can be achieved among users. A CDS contains these coded bits from one block. The formats of a CDS under different conditions are given in Figs. 16.8a–16.8d for the case when a reference sample is used for encoding. For cases when reference sample is not needed, it will not be included in the coded bitstream, and the CDS fields in Fig. 16.8 which contain $J - 1$ will contain J instead.

For the Second-Extension option, in the case when a reference is inserted, a “0” sample is added in front of the $J - 1$ preprocessed samples, so $J/2$ samples are produced after the transformation. When no reference sample is needed, then the n -bit reference field is not used.

16.6 DECODING

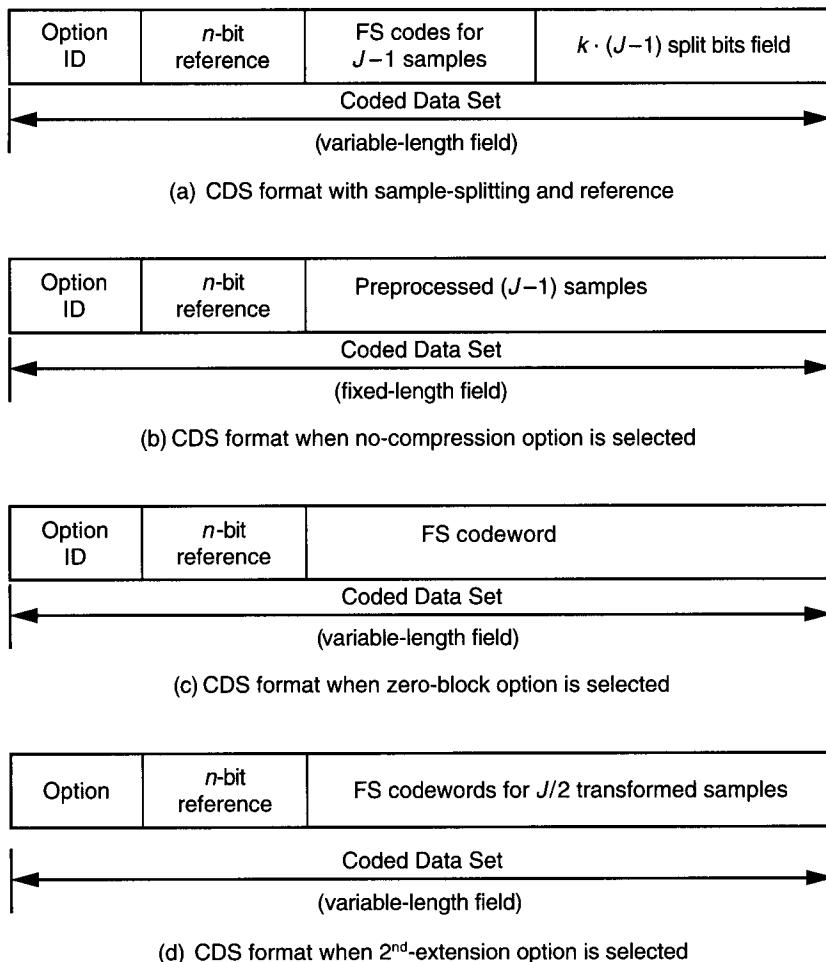
The lossless decoder consists of two separate functional parts, the postprocessor and the adaptive entropy decoder, as shown in Fig. 16.9. The postprocessor performs both the inverse prediction operation and the inverse of the standard mapper operation. The first step toward decoding is to set up the configuration parameters so that both the encoder and the decoder operate in the same mode. These configuration parameters are mission/application specific and are either known *a priori* or are signaled in the Compression Identification Packet defined in [1] for space missions.

The configuration parameters common to both encoder and decoder are as follows:

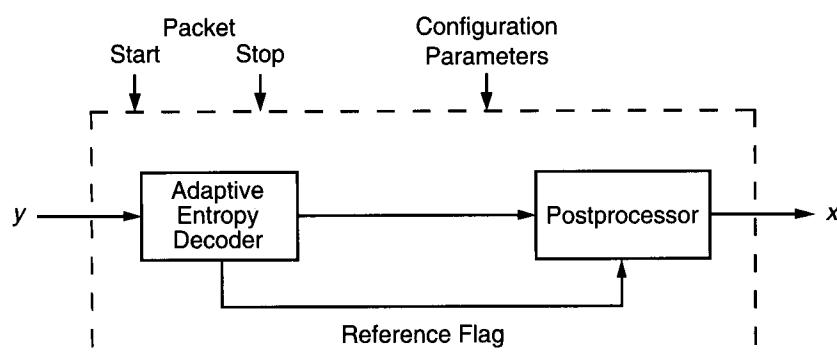
- resolution in bits per sample ($n = 1, \dots, 32$);
- block size in samples ($J = 8$ or 16);
- number of CDSs contained in a packet ($l = 1, \dots, 4096$);
- number of CDSs between references ($r = 1, \dots, 256$);
- predictor type if present;
- sense of digital signal value (positive or bipolar).

For applications such as compressing medical imaging data or image files in a ground archive where packet data structure is not always used for error protection, these decoding parameters should be included in the compressed file as header information. Then Packet Start and Stop in Fig. 16.9 are not needed. The number of CSDs in a scan-line or the number of pixels in a scan-line can be used to decode to the correct number of data samples.

Decoding the coded bits is straightforward once the ID bits are interpreted since the coding is done using FS as the basic structure. Only the second-extension code needs an additional short

**FIGURE 16.8**

Format for a coded data set.

**FIGURE 16.9**

Decoder block diagram.

Table 16.5 Decoding Logic for the Second-Extension Option

m	β	m_s
0	0	0
1, 2	1	1
3, 4, 5	2	3
6, ..., 9	3	6
10, ..., 14	4	10
15, ..., 20	5	15
21, ..., 27	6	21
28, ..., 35	7	28

lookup table to replace computation. For this option, $J/2$ FS codewords will be extracted first. These are then decoded using the following steps:

- Step 1. Obtain m by counting the 0's of the FS codeword.
- Step 2. Obtain β and m_s using Table 16.5 logic.
- Step 3. Calculate $\delta_{i+1} = m - m_s$.
- Step 4. Calculate $\delta_i = \beta - \delta_{i+1}$.

For the Zero-Block option, the single FS codeword following the ID and reference indicates the number of all-zeros blocks or the ROS condition listed in Table 16.3. Using this table, an appropriate number of all-zeros blocks are generated as input to the postprocessor.

The postprocessor reverses the mapper function, given the predicted value \hat{x}_i . When the pre-processor is not used during encoding, the postprocessor is bypassed as well. The inverse mapper function can be expressed as

$$\begin{aligned} \text{if } \delta_i \leq 2\theta, \quad \Delta_i &= \begin{cases} \delta_i/2 & \text{when } \delta_i \text{ is even} \\ -(\delta_i + 1)/2 & \text{when } \delta_i \text{ is odd} \end{cases} \\ \text{if } \delta_i > 2\theta, \quad \Delta_i &= \begin{cases} \delta_i - \theta & \text{when } \theta = \hat{x}_i - x_{\min} \\ \theta - \delta_i & \text{when } \theta = x_{\max} - \hat{x}_i, \end{cases} \end{aligned}$$

where $\theta = \min(\hat{x}_i - x_{\min}, x_{\max} - \hat{x}_i)$.

There will be J (or $J - 1$ for a CDS with a reference sample) prediction error values $\Delta_1, \Delta_2, \dots, \Delta_J$ generated from one CDS. These values will be used in the postprocessor to recover the J -sample values x_1, x_2, \dots, x_J .

To decode packets that may include fill bits, two pieces of information must be communicated to the decoder. First, the decoder must know the block size, J , and the number of CDSs in a packet, l . Second, the decoder must know when the data field in the packet begins and when it ends. From these two pieces of information, the number of fill bits at the end of the data field can be determined and discarded.

The number of valid samples in the last CDS in the packet may be less than J . If so, the user will have added fill samples to the last CDS to make a full J -sample block, and the user must extract the valid samples from the last J -sample block. It is foreseeable that an implementation can be done with variable size for the last block, but that would be outside the scope of the current CCSDS recommendation.

16.7 TESTING

A set of test data for a different data dynamic range, n , has been created. This test set can be used to verify the encoder implementation. This limited set of test vectors can be obtained from the software that is downloadable from the CCSDS Panel-1 home page: http://www.ccsds.org/ccsds/p1_home.html. This data set will cause all the split-sample options, the no-compression option, and the Second-Extension option to be selected once. The Zero-Block option will be selected at least once for up to $n = 14$ when only 256 data points are requested. If the user allows more than 256 test data points to be written, then the Zero-Block option will be exercised for all n up to $n = 16$. Current test programs generate test data for resolution from $n = 4$ to $n = 16$ bits.

16.8 IMPLEMENTATION ISSUES AND APPLICATIONS

For space applications, several systems issues relating to embedding data compression on board a spacecraft have been addressed. These include onboard packetization for error containment, buffering for constant but bandwidth-limited downlink rate, sensor response calibration, and compression direction for optimized compression performance [2].

For applications on ground, normally the requirement is much more relaxed; most applications do not use any packet data structure with added channel coding to protect from transmission or storage bit errors. There is also no need to impose a limit on buffering either input or compressed data. To achieve the best performance, users only have to engineer a preprocessor that would decorrelate the input signal into a non-negative integer signal that resembles a Laplace distribution. In certain situations the knowledge of how data are acquired from various sensors and formatted can be extremely useful in optimizing compression performance. An example is data collected from an array of different types of sensors (e.g., strain gauge, thermometer, accelerometer, fuel gauge, pressure gauge) from spacecraft or any other machinery. This data is commonly arranged in sets in the sequence of time steps T_i when data is collected, shown in Fig. 16.10.

It is customary to first try to apply a compression algorithm directly on the time sequence of the data as it is collected in such manner. The resulting poor performance can be attributed to the different characteristics of these sensors. To optimize performance, compression should be applied to data collected from each sensor in a time sequence independent of other sensors as shown in Fig. 16.11.

In an effort to broaden the domain of applications, this algorithm has been applied to processed science data such as sea surface temperature and vegetation index in floating-point representations and achieved over 2:1 lossless compression as well. This can be accomplished by judiciously executing compression in the direction shown in Fig. 16.12. In this scenario, the preprocessor will

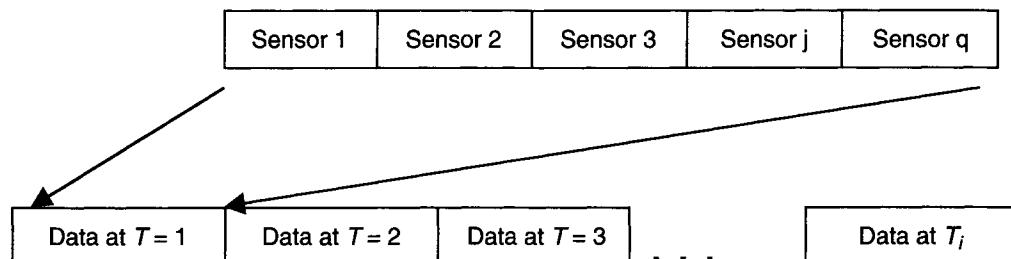


FIGURE 16.10

Common sensor data arrangement at different time steps.

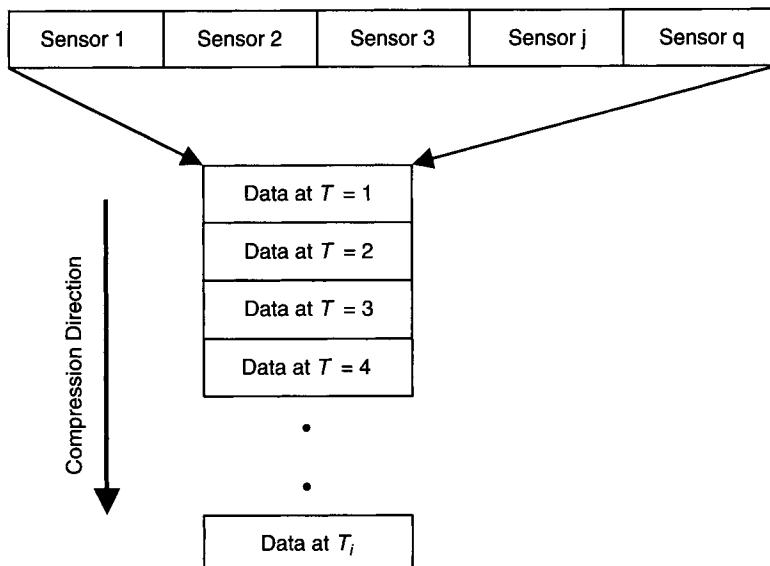


FIGURE 16.11
Optimal compression scheme.

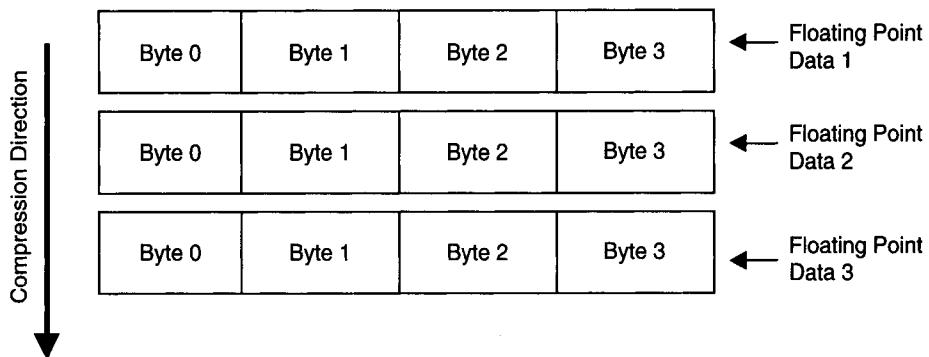


FIGURE 16.12
Compressing floating-point science data.

take the 4 bytes from one floating-point number as if they were from four different data sources. Each data source now has data of 1-byte (value 0–255) dynamic range. A unit delay predictor is then used before the entropy coder.

The scheme in Fig. 16.12 can be implemented with four buffer arrays, each holding the i th byte (i being $0, \dots, 3$) from the floating-point data. The length of the array can be fixed at the length corresponding to one scan-line or any number of J -sample blocks. The compressed bits will then be concatenated at the end of compressing each array. For double precision or complex-valued science data with 8 bytes per data point, the same scheme can be used.

The CCSDS e_Rice compression scheme has been utilized on medical imaging data with good results [10]. Applications also have found their way into science data archives, seismic data, acoustic data, and high-energy particle physics as well.

16.9 ADDITIONAL INFORMATION

1. One source of available high-speed integrated circuits and software for the e_Rice algorithm is www.cambr.uidaho.edu.
2. The Second-Extension option in the e_Rice algorithm is protected by U.S. Patent 5,448,642 assigned to NASA. For implementation for commercial usage, contact the technology commercialization office at the Goddard Space Flight Center in Greenbelt, Maryland.

DEDICATION

The author dedicates this chapter to the memory of Warner H. Miller, whose vision initiated the technology development of the e_Rice algorithm and whose encouragement helped achieve its completion.

16.10 REFERENCES

1. *Lossless Data Compression*. 1997. Consultative Committee for Space Data Systems CCSDS 121.0-B-1 Blue Book, May 1997.
2. *Lossless Data Compression*. 1997. Consultative Committee for Space Data Systems CCSDS 120.0-G-1 Green Book, May 1997.
3. Huffman, D. A., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, Vol. 40, pp. 1098–1101.
4. Welch, T. A., 1984. A technique for high-performance data compression. *IEEE Computer*, Vol. 17, No. 6, pp. 8–19, June 1984.
5. Witten, I. H., R. M. Neal, and J. G. Cleary, 1987. Arithmetic coding for data compression. *Communications of the ACM*, Vol. 30, No. 6, pp. 520–540. June 1987.
6. Rice, R. F., 1979. Practical universal noiseless coding. In *Proceedings of the SPIE Symposium*, Vol. 207, San Diego, CA, August 1979.
7. Yeh, P.-S., R. F. Rice, and W. H. Miller, 1993. On the optimality of a universal noiseless coder. In *Proceedings of the AIAA Computing in Aerospace 9 Conference*, San Diego, CA, October 1993.
8. Venbrux, J., P.-S. Yeh, and M. N. Liu, 1992. A VLSI chip set for high-speed lossless data compression. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, No. 4, December 1992.
9. Yeh, P.-S., R. F. Rice, and W. H. Miller, 1991. *On the Optimality of Code Options for a Universal Noiseless Coder*, NASA/JPL Publication 91-2, February 1991.
10. Venbrux, J., P.-S. Yeh, G. Zweigle, and J. Vessel, 1994. A VLSI chip solution for lossless medical imagery compression.” In *Proceedings of the SPIE’s Medical Imaging 1994*, Newport Beach, CA.

Lossless Bilevel Image Compression

MICHAEL W. HOFFMAN

17.1 BILEVEL IMAGE COMPRESSION

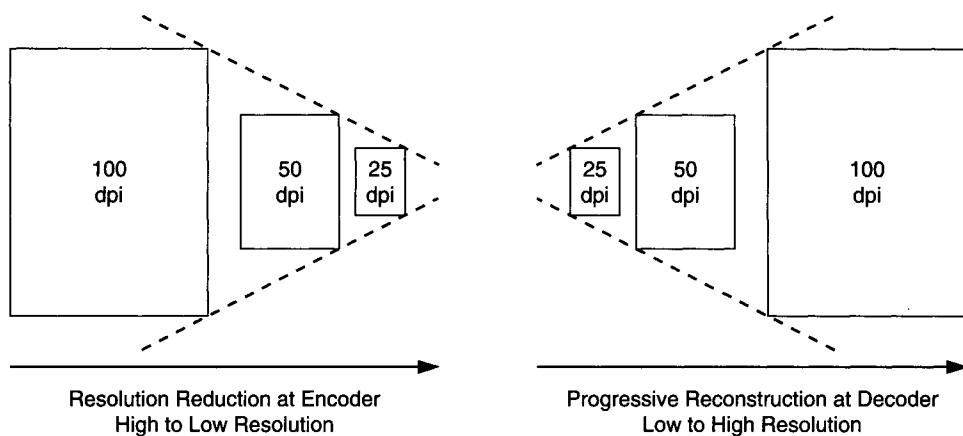
This chapter describes two international standards for lossless bilevel image compression that are commonly referred to as “JBIG” and “JBIG2.” JBIG2 also allows lossy compression of bilevel images as a trade for greater compression in areas of documents in which a lossless reproduction may not be warranted, such as a halftone background or fill pattern. The JBIG standard is described in the next section. In Section 17.3 the JBIG2 standard is described. Finally, the performance results of current lossless bilevel compression standards are compared using published reports.

17.2 JBIG

ITU-T Recommendation T.82 [3] specifies the eponymous JBIG standard that was drafted by the Joint Bilevel Image Experts Group (JBIG). This standard provides an alternative to Recommendations T.4 and T.6 in providing lossless compressed representations of bilevel images for more efficient transmission and storage of these images.

17.2.1 Overview of JBIG Encoding/Decoding

While it can be used effectively on grayscale or color images, JBIG was primarily designed for the compression of bilevel images, i.e., those images that have two colors: background and foreground. The primary applications for JBIG compression include facsimile transmission, document storage, and document distribution. These documents may contain text, line drawings, and halftone images. This variety of applications requires different modes of decomposing, transmitting, and reconstructing the bilevel images. In a facsimile transmission, for example, there may be little

**FIGURE 17.1**

Resolution reduction and progressive encoding and reconstruction.

need for a progressive reconstruction of the image. In this case, a sequential transmission (left to right, top to bottom) of the original image pixel data may be preferred. If documents are being archived for later retrieval and distribution over networks, however, a progressive reconstruction may significantly reduce the network resources or operator effort required to search for a specific document or type of document. Other considerations, such as the memory required to build up progressive approximations before printing a full-sized image, also impact the decisions on how to store, access, or transmit the image data.

Figure 17.1 illustrates the resolution reduction and progressive reconstruction used in JBIG progressive transmission. At the encoder, the order of processing is from the original and highest resolution image to the lowest resolution image. The decoder reconstructs from the lowest resolution image first, successively refining the image (and increasing the resolution) until the original image is recovered. In Fig. 17.1 the lower resolution images are shown as smaller in size than their higher resolution counterparts to indicate that there are fewer data bits in a low-resolution image. Of course, if these different resolution images were rendered onto a typical facsimile page, the size of these rendered images would be the same. The number of resolution reductions is variable—a selection that results in a final resolution of 10 to 25 dots per inch (dpi) allows for useful thumbnail representations. Original image source resolutions may be 200 to 400 dpi for facsimile applications or 600 dpi or higher for documents destined for laser printing. In a practical sense, the standard imposes very few restrictions on these basic parameters.

JBIG allows for a variety of operational modes. Figure 17.2 shows the relationship among three distinct modes of operation. Single-progression sequential mode means that the pixels are scanned in raster order (left to right, top to bottom) and encoded for the entire image. Progressive mode and progressive-compatible sequential mode both use identical progressive reconstructions of the image. However, the ordering of the data is different in the two modes. The progressive mode reconstruction occurs for the entire image at one resolution. Reconstructions for subsequent resolutions are also made over the entire image.

An image can be broken into horizontal stripes and these stripes can be (loosely) treated as separate images. Progressive-compatible sequential mode performs the progressive refinement over a stripe of the image until that stripe is reconstructed at the highest resolution and then continues with the next stripe of the image. These image stripes are depicted in Fig. 17.3. The standard allows the number of lines in a stripe at the lowest resolution to be a free parameter. Note that the stripe is required to run the full width of the image. If an odd number of lines are

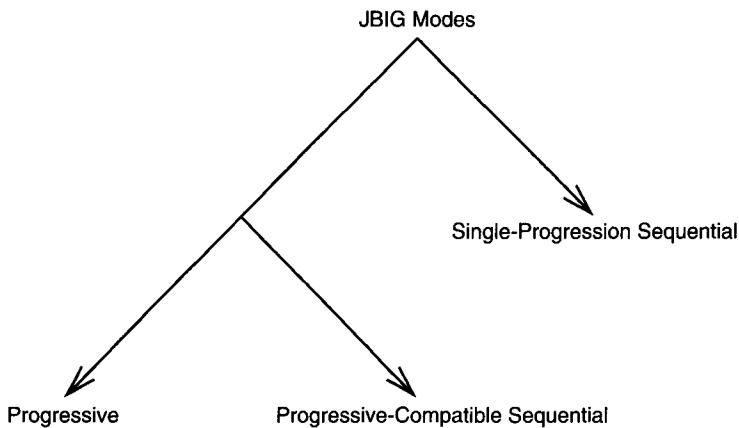


FIGURE 17.2
JBIG modes of operation.

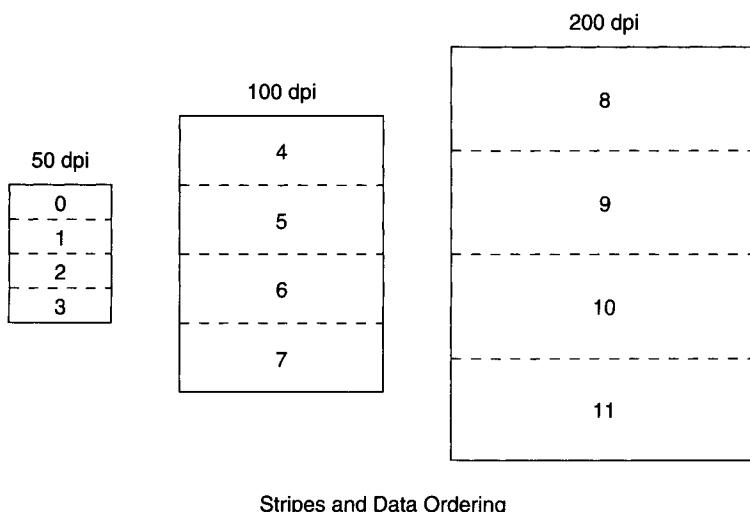


FIGURE 17.3
Assignment of stripes within various resolutions of a given image and the possible data orderings for a single bit-plane.

included at a stripe in a higher resolution, the final line in the stripe needs to be replicated before resolution reduction.

Figure 17.3 also indicates various choices for ordering the progressive stripe data. The binary parameter SEQ allows either progressive full image data ordering (SEQ = 0) or progressive-compatible sequential ordering (SEQ = 1). The binary parameter HITOLO determines whether the data is ordered in highest to lowest resolution, the order in which it is processed at the encoder (HITOLO = 1), or in lowest to highest resolution, the order in which it is processed at the decoder (HITOLO = 0). Note that either the decoder or encoder is burdened with storing the bulk of the representation—this choice, however, can be made depending on the needs of a particular application. As mentioned previously, note that the compressed data transmitted is the same for either the progressive or progressive-compatible sequential modes; however, the order in which it is transmitted is different. Finally, if more than a single bit-plane is being considered, there are other additional parameters for determining the ordering of the stripes within each resolution and bit-plane.

17.2.2 JBIG Encoding

This section describes the encoding details of JBIG. JBIG encoding includes the following components: resolution reduction, prediction, model templates, and encoding. Resolution reduction results in a lower resolution image as well as data that needs to be encoded so that reconstruction of the current resolution is possible given the lower resolution image. Prediction attempts to construct the higher resolution pixels from the lower resolution pixels and neighboring higher resolution pixels that are available to both the encoder and the decoder. Two basic types of prediction are used: typical prediction and deterministic prediction. Neighboring pixels are used to provide a template or context for more efficient entropy coding. Both fixed templates and adaptive templates are used. Finally, a context-based adaptive arithmetic encoder is used to encode those pixels that cannot be predicted and require encoding. Some differences exist between encoding operations at a differential layer and at the bottom layer and these will be described as required.

17.2.2.1 Resolution Reduction

D defines the number of doublings of resolution in an encoding of an image. At the highest resolution there are X_D pixels per row and Y_D rows in the image. R_D is the sampling resolution of the original image (typically stated in dpi). If sequential coding is desired, D is set to 0 and the original image is treated as the bottom layer (i.e., the lowest resolution layer). Figure 17.4

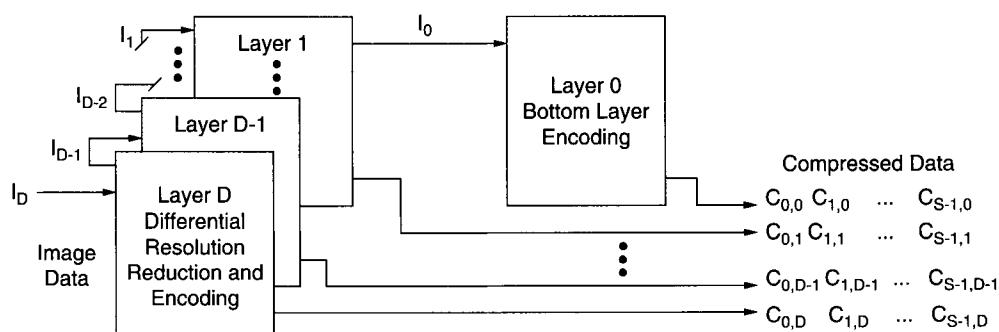
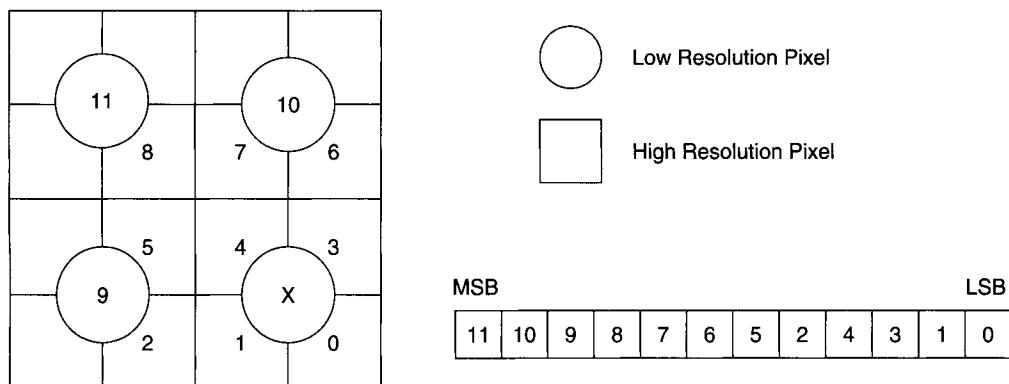


FIGURE 17.4

JBIG encoder resolution reduction and encoding.

**FIGURE 17.5**

JBIG encoder resolution reduction neighborhood.

illustrates the resolution reduction and data generation for a JBIG encoder. The image at resolution $D - k$ is referred to as I_{D-k} ($k = 0, \dots, D$). The coded image data for stripe s ($s = 0, \dots, S - 1$) at resolution $D - k$ is denoted $C_{s,D-k}$. The functional blocks for each differential layer resolution reduction and encoding are identical. The bottom layer encoding contains significant differences from the differential layer encoding. Each differential layer takes as input the image data from the next higher resolution differential layer. In addition, each differential layer performs a resolution reduction and outputs coded data that allows for reconstruction of the input (higher resolution) image given the output (lower resolution) image.

The resolution reduction algorithm consists of a table-based look-up given a neighborhood of surrounding pixels taken from both the high- and low-resolution images. Figure 17.5 illustrates the relationship among the neighboring pixels in both the high- (squares) and low- (circles) resolution images. The pixel to be determined in the reduced resolution image is denoted X , while the pixels numbered 11, 10, ..., 0 form the 12-bit context for pixel X . This context is used as the index into a resolution reduction table with 4096 one-bit entries. The bit from the table for a given context is used as the value for X . While it is possible to define a different resolution reduction table than that included in the JBIG recommendation, it is important to note that this change will require matching tables for deterministic prediction. The goal in designing the table set forth in the standard was to preserve edges and lines via exceptions to a general resolution reduction rule. While not necessarily enhancing compression, these exceptions do enhance the visual information content of the low-resolution images.

In many cases, the pixels needed for a given context may not exist in the original image or the stripe being processed. In those cases the standard provides the following general rules for edges:

- the top, left, and right edges are assumed to be surrounded by the background color (0), and
- the bottom line is assumed to be the replication of the last line in the image (or stripe).

So if either or both of the vertical and horizontal dimensions of the image at level d are odd, additional pixels at the right and bottom are added according to these general rules. In addition, these rules are followed as needed for generating context pixels for edge pixels in the image. When pixels are required across stripe boundaries pixels from stripes above are used from the actual image since these will be available to the decoder, but pixels from below the current stripe are generated by replication from the current stripe even if these pixels exist in the original image.

17.2.2.2 Differential Layer Prediction

The prediction used in the differential and bottom layers is sufficiently different to warrant separate discussions. In differential layer encoding, after resolution reduction there are three possibilities for encoding a high-resolution pixel:

- it can be ignored (not coded) because typical prediction uniquely determines its value,
- it can be ignored (not coded) because deterministic prediction uniquely determines its value, or
- it is encoded by an arithmetic coder based on a context of neighboring pixels (from high- and low-resolution images).

In this section the first two possibilities are discussed.

17.2.2.2.1 Differential Typical Prediction Figure 17.6 illustrates the neighborhood of the eight low-resolution pixels closest to low-resolution pixel X . The four phases of the high-resolution pixels associated with X are also indicated in the figure. A pixel is called “not typical” if it and all eight of its neighboring pixels are the same color, but at least one of the four high-resolution pixels associated with it is not. If a line of low-resolution pixels contains any “not typical” pixels, then the *line* is referred to as not typical. For each pair of high-resolution lines, a fictitious pixel called LNTP (Line Not Typical) is inserted to the left of the first pixel in the first of the two-line set of high-resolution pixels. LNTP is set to zero unless a low-resolution line is found to be “nontypical.” In that case this pixel is set to 1. A particular context that is rarely used in coding normal bilevel images is reused for arithmetic encoding of the LNTP pixel.

When a low-resolution pixel is found to be “typical” in a line deemed typical, then the four phases of the high-resolution pixels associated with that pixel are not encoded. The introduction to the JBIG recommendation states that on text or line-art images, typical differential prediction allows coding to be avoided for approximately 95% of the pixels. This allows a considerable reduction in execution time since no other processing is required when typical prediction is used. The overhead for using typical prediction includes sending the extra pixel LNTP for every other

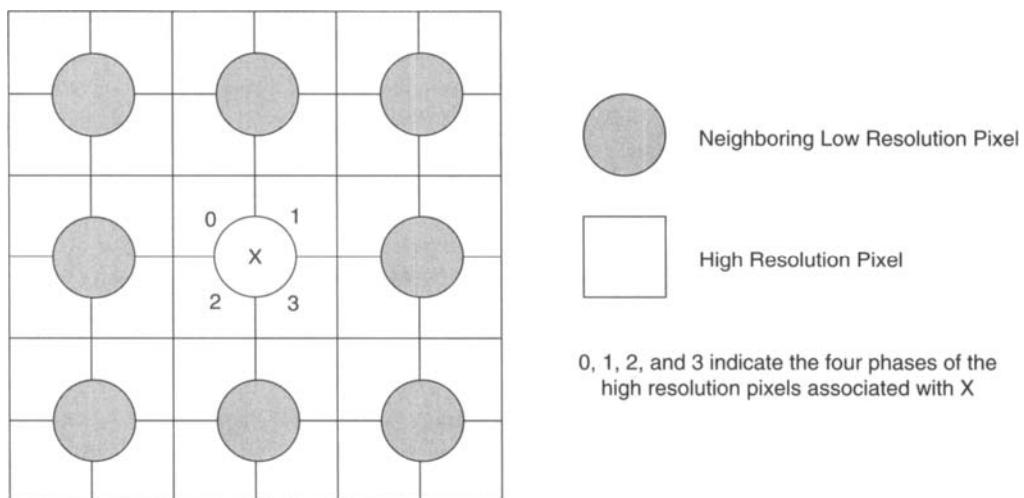
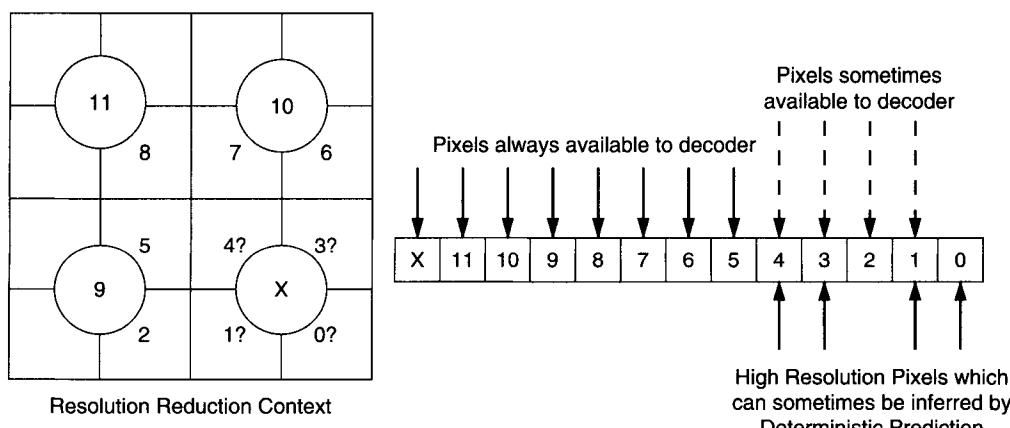


FIGURE 17.6

The neighborhood of the eight low-resolution pixels closest to X and the four phases of the high-resolution pixels associated with X .

row of the high-resolution image. Note that at the decoder the lowest levels are reconstructed first. This allows determination of whether a given low-resolution pixel can be typical at the decoder. When the line in a low-resolution image is at a stripe boundary, the general rules for edge pixels are used to replicate the last row in the stripe even though the actual pixel may be available at the decoder. Progressive-compatible sequential encoding, for example, would not have this pixel available, while progressive encoding would—but the edge pixel rules are used in both cases. Finally, note that typical prediction is optional for differential layers.

17.2.2.2 Deterministic Prediction (differential layer only). Since the table look-up for resolution reduction sometimes results in a high-resolution pixel being determined by the information already available to the decoder, there is no need to encode this pixel. In JBIG this is referred to as deterministic prediction. Deterministic prediction is available only for differential layer encoding. Figure 17.7 illustrates the relationship between the table-based resolution reduction algorithm and deterministic prediction. Given a resolution reduction table (such as the default table), there are patterns of bits $X, 11, 10, 9, \dots$ available to the decoder that can be used to predict the values of the four phases of the high-resolution pixels associated with X . These phases are bit positions 4, 3, 1, and 0 from the resolution reduction context. For each of these pixels there is a chance that the particular sequence of bits in the context already available to the decoder removes any ambiguity about the pixel and obviates the need for coding that pixel. The number of deterministic hits for each of the four phases (using the default resolution reduction tables) is indicated in a table provided in the figure. The standard recommendation indicates that using deterministic prediction allows additional compression of about 7% on the bilevel images in the test set.



Phase	Pixel	Number of Contexts	Number of Deterministic Hits with Default Resolution Reduction Tables
0	4	256	20
1	3	512	108
2	1	2048	526
3	0	4096	1044

FIGURE 17.7

Resolution reduction context and its relationship to deterministic prediction.

Deterministic prediction for the differential layers is optional in JBIG. In addition, should a different resolution reduction table be used for a particular application, then either deterministic prediction should not be used or the deterministic prediction table must be matched to the particular resolution reduction algorithm used. This deterministic prediction table needs to be included in the JBIG header if deterministic prediction is to be used.

17.2.2.3 Bottom Layer Typical Prediction

For the bottom layer image data only typical prediction is available. (Note that if the sequential encoding mode is used, then this bottom layer image is simply the original image.) As with the differential layer prediction, bottom layer typical prediction is an option. The i th row (or line) in the bottom layer image data is said to be not typical if it differs in any pixel from the line above it. If it is not typical, then $LNTP_i$ is set to 1. A fictitious pixel called $SLNTP$ ("silent pea") is generated as

$$SLNTP_i = !(LNTP_i \oplus LNTP_{i-1}), \quad (17.1)$$

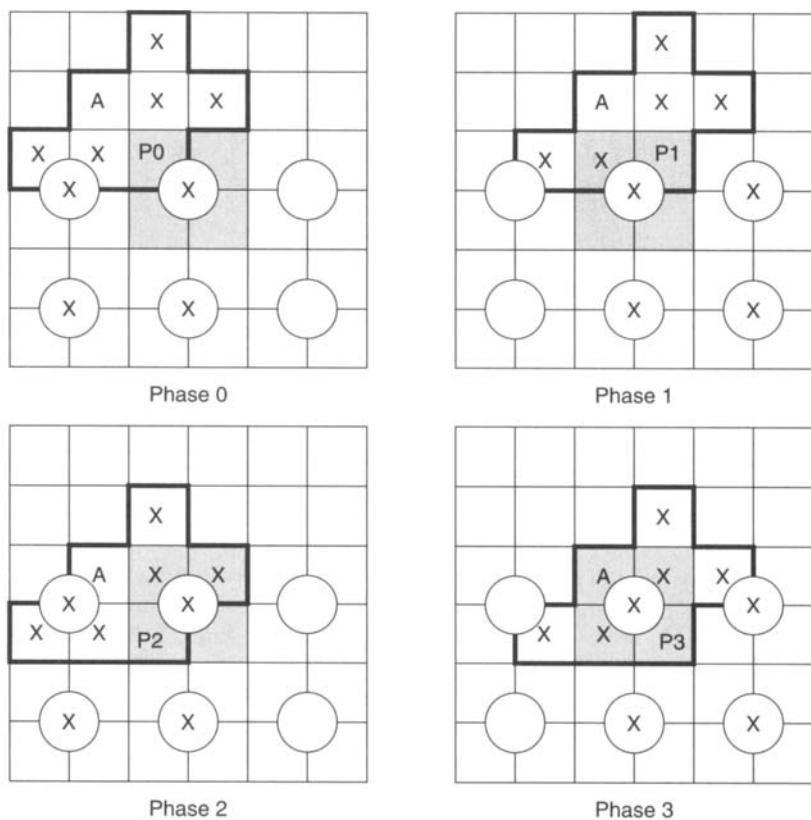
where \oplus indicates exclusive-or and $!$ indicates logical negation. For the top line, $LNTP_{-1}$ is set to 1. The virtual pixel location for sameness indicator $SLNTP_i$ is to the left of the first pixel in line i . Note that for the bottom layer, the virtual pixel occurs in every line when typical prediction is used.

17.2.2.4 Model Templates

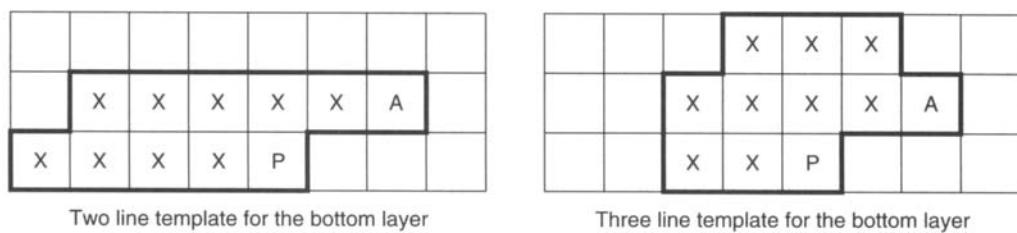
When typical or differential prediction is used, the predictors generate a 0 or 1 (for the value of the pixel being predicted) or a 2 to indicate that this particular form of prediction cannot be applied. Note that this output does not need to be transmitted since the prediction is designed to be performed in an identical fashion at both the encoder and decoder. When prediction cannot be used, then the pixel in question needs to be encoded. Context-based adaptive arithmetic coding is used to encode these pixels. In this section the templates used to generate the contexts are described.

17.2.2.4.1 Differential Layer Templates Figure 17.8 depicts the templates used for differential layer encoding of the four phases of the high-resolution pixels. In the figure, pixels labeled X or A are the pixels used to form the context for coding the pixel starting with P . In addition to the 10 bits representing these pixels, 2 bits are required to identify the coder used for each of the four phases of high-resolution pixels. This results in 4096 possible contexts (one of which is also used to code $LNTP$).

The X pixel locations are fixed in the positions indicated in Fig. 17.8. The A pixel denotes the adaptive part of the template or, simply, the adaptive template. This pixel can be moved to a wide range of locations to allow better coding for repetitive regions such as may occur in half-toned computer generated images. The range of motion is $\pm M_x$ in the horizontal direction and from 0 to M_y in the vertical direction. M_x can be set up to a maximum of 127 while M_y can be set up to a maximum of 255. These two parameters limit the range of values that can be used for a given bilevel image. The actual values can be varied within a stripe of data by using an escape sequence followed by the line in which the change occurs and the value of the horizontal and vertical offsets for the adaptive template. Up to four moves per stripe are allowed, with the order of the move escape sequences corresponding to the increasing number of lines in which the moves become effective. Setting M_x and M_y to zero for an image indicates that the default position will be the only location for the adaptive template.

**FIGURE 17.8**

Differential layer templates for each of the four phases of the high-resolution pixels.

**FIGURE 17.9**

The two- and three-line bottom layer templates.

17.2.2.4.2 Bottom Layer Templates Figure 17.9 illustrates the two- and three-line templates used for the bottom layer. As with the previous figure for the differential layer templates, the X pixels are set in a fixed location within the template and the A pixel denotes the adaptive template. The restrictions on the location of the adaptive template are identical to those for the differential layer adaptive template described previously. As is the case with the differential layer adaptive template, a location specification of $(0, 0)$ indicates the default location indicated in the figure. Note that only 10 bits specify the context for the bottom layer templates. This means a total of 1024 contexts for the arithmetic encoder for either the two- or three-line contexts.

Note that the order in which the template bits are arranged to form the context is unimportant, since it is only the state of the arithmetic coder that needs to be tracked—and this is done in an identical fashion for the encoder and the decoder regardless of how the template bits are ordered to form the context index. As long as the formation of the context is consistent throughout the encoding (decoding) process, it is irrelevant that it is done in a manner different than that used in the decoding (encoding) process. This is true for both the bottom layer and differential layer encoding.

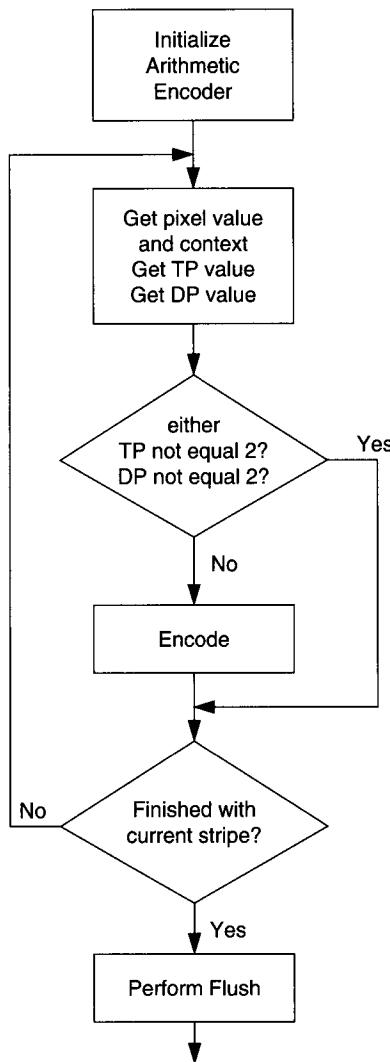
17.2.2.5 Adaptive Arithmetic Coding

For each pixel being encoded the values generated by typical prediction (TP) and deterministic prediction (DP) are used to decide whether arithmetic coding is needed. If either prediction mode is not being used, the corresponding value for this predictor is always set to 2. When either prediction provides the value for the current pixel, then this value (0 or 1) indicates that no arithmetic encoding is needed. If arithmetic encoding is required (both TP and DP equal 2), then the context and the value of the current pixel are used by the adaptive arithmetic encoder. The top level flow diagram for JBIG adaptive arithmetic encoding is shown in Fig. 17.10.

The actual encoding depicted in the figure is a variant of the Q-coder adaptive binary arithmetic coder. The coder tracks a most probable symbol (MPS) and a least probable symbol (LPS) and applies a multiply-free set of update equations (see [1], for example). For each context there is a 1-bit value of the MPS and a 7-bit value indicating 1 of 113 possible states for the coder. As pixels are encoded, the arithmetic coding interval is updated and, if required, a renormalization (i.e., a set of doublings) of the interval occurs. This renormalization process generates the bits that form the coded sequence. After each renormalization process a probability estimation process is undertaken to reset the state of the coder for a given context. It is possible that the MPS switches during encoding—certain states of the encoder are associated with this switch and the table look-up for the state indicates when a switch occurs. At the end of a stripe of data, a flush routine removes unnecessary all 0 bytes from the datastream. These will be generated within the decoder as required to allow for complete decoding of the coded bitstream.

17.2.3 Data Structure and Formatting

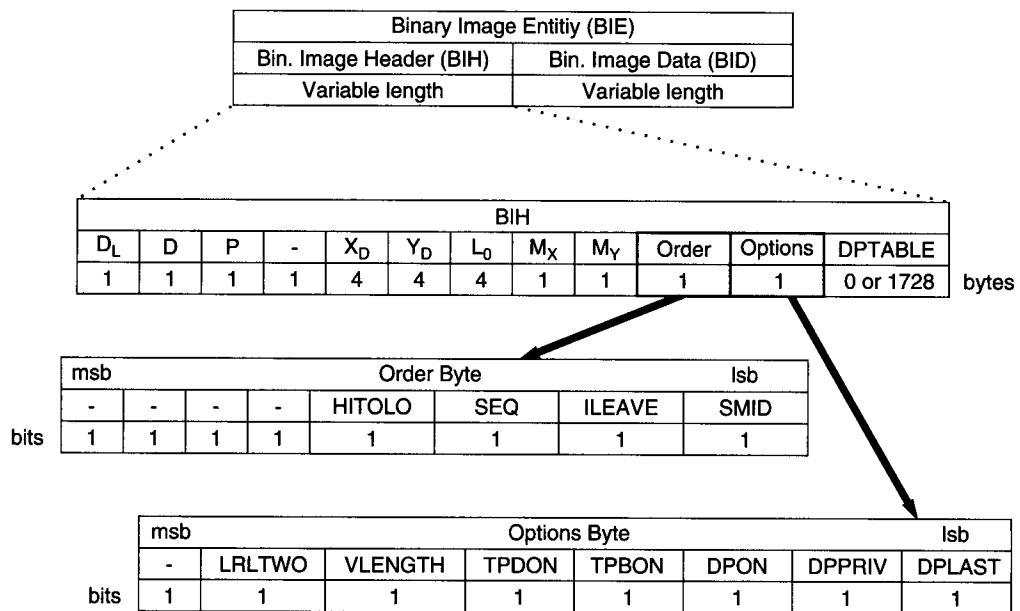
The fundamental unit of JBIG image data is the Bilevel Image Entity (BIE). The BIE is composed of a Bilevel Image Header (BIH) and Bilevel Image Data (BID). It is possible, but not required, to use multiple BIEs for a single image. Figure 17.11 illustrates the relationship of the BIH and BID within the BIE. In addition, the figure breaks out the components of the BIH with labels and sizes of key elements. The BIH contains the initial resolution layer, D_L , within the BIE and the final resolution layer, D , within the BIE. The value P specifies the number of bit-planes in the image (P is 1 for bilevel images). X_D and Y_D are the size of the image at its *highest* resolution. These 4-byte entries are encoded with most significant byte first and allow for image sizes of $2^{32} - 1$ (or over 4×10^9) pixels per dimension. L_0 specifies the number of lines in a stripe at the lowest resolution (this is also a 4-byte entry coded most significant byte first). M_x and M_y are 1-byte entries specifying the maximum allowed range of motion for the adaptive template in the BIE. The *Order* byte contains the 1-bit values *HITOLO* and *SEQ* that specify the order in which the stripes and bilevel data are presented within the BIE. *ILEAVE* and *SMID* perform similar functions when the image consists of multiple bit-planes—these are used with *HITOLO* and *SEQ* to provide up to 12 different orderings of data segments. In the *Options* byte, the *LRLTWO* bit determines whether the two-line template (one) or three-line template (zero) is used in encoding the lowest resolution data. The *VLENGTH* bit allows a redefinition of the image length to occur. *TPDON* indicates that typical prediction in the differential layer is used when it is set to 1 while

**FIGURE 17.10**

Flow chart for the adaptive arithmetic encoding performed in JBIG.

TPBON performs a similar function for bottom layer typical prediction. *DPON* indicates that the deterministic prediction is used (recall that this affects only differential layer image data). The *DPPRIV* bit indicates that a private resolution reduction algorithm has been used and it specifies that a private deterministic prediction table needs to be used when it is set to 1. The *DPLAST* bit indicates whether to use the previously loaded private deterministic prediction table (one) or to load a new DP table (zero). Finally, if a private deterministic prediction table needs to be loaded the 1728 bytes of data in this table follow the *Options* byte.

The BID is composed of floating marker segments followed by Stripe Data Entities (SDE). An escape byte, ESC, is used to indicate marker codes. These marker codes can be used to terminate Protected Stripe Coded Data (PSCD), to abort a BID prematurely, or to indicate a floating marker segment. The floating marker segments include an adaptive template move segment, a redefinition of image length segment, a comment segment, and a reserved marker byte. The adaptive

**FIGURE 17.11**

JBIG data structure and header information.

template move segment specifies the line at which the change occurs and the positions (horizontal and vertical) to which the adaptive template is moved. The new length segment specifies the new length of the image. To avoid improper operation when the arithmetic coder binary data produces a byte-aligned ESC symbol, the Stripe Coded Data (SCD) is searched and the occurrences of byte-aligned ESC symbols are replaced with the ESC symbol followed by a STUFF byte. This produces the PSCD that is included in the SDE within the BID. There can be an arbitrary number of SDEs in a BID. Floating marker segments occur between SDEs within the BID.

17.2.4 JBIG Decoding

JBIG decoding is a fairly straightforward reversal of the steps used in the encoder. The decoder starts with the coded data from the lowest level image data and reconstructs this bottom layer image. If sequential encoding was used, the decoder is finished. If progressive coding was used, the decoder then undoes the resolution reduction by reconstructing the high-level image from the low-level image and the coded image data for that resolution layer.

The adaptive arithmetic decoding mirrors the adaptive process used by the encoder. If typical prediction and deterministic prediction are used, then the JBIG decoder is required to detect situations in which the encoder would have avoided encoding a given pixel and use its own local prediction in these cases. The determination of whether particular lines are typical is made by the encoder and this information is transmitted as the first pixel in each line (bottom layer) or in every other line (differential layer). As with the encoder, the operation of the differential layers at the decoder is identical. One difference is that if the bilevel image is to be rendered, the final differential decoding can be sent directly to the display device rather than being stored. Note that the progressive-compatible sequential mode can be used to avoid the memory issues involved in reconstructing large images by defining stripes to be a manageable number of lines.

To determine the number of lines, lines per stripe, or pixels per line in a decoded differential layer, the decoder uses the actual number of rows and columns in the highest resolution image, the number of lines per stripe at the lowest layer, and the algorithm for resolution reduction to reconstruct the correct number of lines. These data are included in the BIE header information. For example, if the number of pixels per line at resolution d is 21, then it is possible for the number of pixels per line at resolution $d + 1$ to be 41 or 42. While one may be tempted to say “the correct answer is 42,” it is possible that the number of pixels per line is 41, as would occur if the original image contained 2561 pixels per line.

According to the JBIG recommendation, the best compression is typically obtained using the sequential mode of operation. Based on implementation and use considerations, the progressive mode or progressive-compatible sequential mode may be more appropriate for a given application. If a particular image is to be compressed for use on a variety of platforms, progressive operation allows different users to meet their fidelity requirements and produce a version of the document without undue computation or delay. Recall that the only difference between progressive and progressive-compatible sequential coding is the order in which the same coded data is organized and transmitted.

17.3 JBIG2

ITU-T Recommendation T.88 [4] specifies the standard commonly referred to as JBIG2. This recommendation was drafted by the Joint Bilevel Image Experts Group committee, the same group that drafted the JBIG recommendation. This recently (2000) approved standard provides an alternative to Recommendation T.82 (JBIG) [3] and Recommendations T.4 and T.6 in providing lossless compressed representations of bilevel images for more efficient transmission and storage of these images. In addition, JBIG2 provides for lossy compression of bilevel images. The amount and effect of this lossy compression are controlled during encoding and are not specified as part of the standard. Generally, lossy compression of halftone regions of bilevel images can provide dramatic reductions in the compressed image size in exchange for small reductions in quality.

17.3.1 Overview of JBIG2

JBIG2 allows bilevel documents to be divided into three parts: text regions, halftone regions, and generic regions. Text regions consist primarily of symbols (letters, numbers, etc.) that are relatively small, such as the standard font sizes used for document preparation. These symbols are aligned in either a left-right or top-bottom row format. JBIG2 allows the formation of arbitrary symbol dictionaries. These dictionaries form the basis of the compressed representations of the symbols. The fact that particular symbols are mapped to a finite set of dictionary symbols indicates a certain lossy quality to the representations for symbol occurrences. Hence, a refinement step for generating the final symbol from the initial estimate provided by the dictionary entry is included as an option.

Halftone regions consist of regularly occurring halftone patterns that allow bilevel images to be used for representing various shades and fill patterns, as well as grayscale images. In a halftone region a particular pattern dictionary is used to represent small sections of the bilevel image. The indices of the patterns in the dictionary that are matched to the original image are treated as a pseudo-grayscale image at a reduced spatial resolution from the bilevel image. This pseudo-grayscale image is encoded in bit-planes and transmitted to the decoder along with angular orientation and grid size information to provide for a lossy representation of a halftone portion of a bilevel image.

Generic regions consist of line drawings, large symbols, or other bilevel components that have not been identified or encoded as halftone or text regions. In addition, the algorithms used for compressing generic regions are the basis of the representations for the symbol dictionary components, the pattern dictionary components, and the bit-planes of the pseudo-grayscale images used in the halftone regions. Generic region coding also allows refinements to be applied to other regions. This permits *quality progressive* representations to be encoded.

Note that, in contrast to JBIG which treated each page or page stripe as a single image, JBIG2 is focused on compressing documents containing different types of content and multiple pages with similar components. To this end, JBIG2 is a segment-based encoding scheme, compared to JBIG, which is an image-based encoding scheme. Segments of a document can contain one of the basic region types mentioned previously, dictionaries for text or patterns, general page layout information, or decoding control information. The ordering of the segments is quite flexible and allows both quality progressive and *content progressive* representations of a document. In a content progressive representation, perhaps text segments would be displayed first, followed by line art and, finally, halftone background segments. These types of representations are particularly useful for network distribution of documents. Note that the rectangular regions defined as text, halftone, or generic can overlap and there can be virtually any number of these per page. Intermediate results are stored in auxiliary buffers that are combined to form the final page buffer. Figure 17.12 illustrates a bitmap, one of the basic building blocks used within JBIG2 regions. The overall structure of the JBIG2 file can be sequential, in which each segment header is immediately followed by the data header and the data for that segment, or random access, in which the segment headers are concatenated at the beginning and the header data and data for each segment follow in the same order as the segment headers.

Whereas the basic compression technology in JBIG was context-based arithmetic coding, JBIG2 allows either context-based arithmetic coding (MQ coding similar to JBIG) or run-length encoding (Modified Modified Relative Element Address Designate (READ) (MMR) coding similar to that used in the Group 4 facsimile standard, ITU-T.6). The primary advantage of MQ coding is superior compression. The primary advantage of MMR coding is very fast decoding.

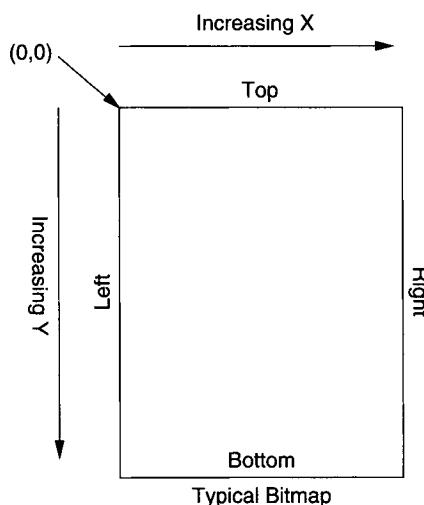


FIGURE 17.12

JBIG2 bitmap: four edges with (0,0) referring to the top left corner. Bitmaps can be given one of four orientations within a JBIG2 region.

Depending on the application, either of these alternatives may be more appealing. MMR coding is described in Chapter 20.

17.3.2 JBIG2 Decoding Procedures

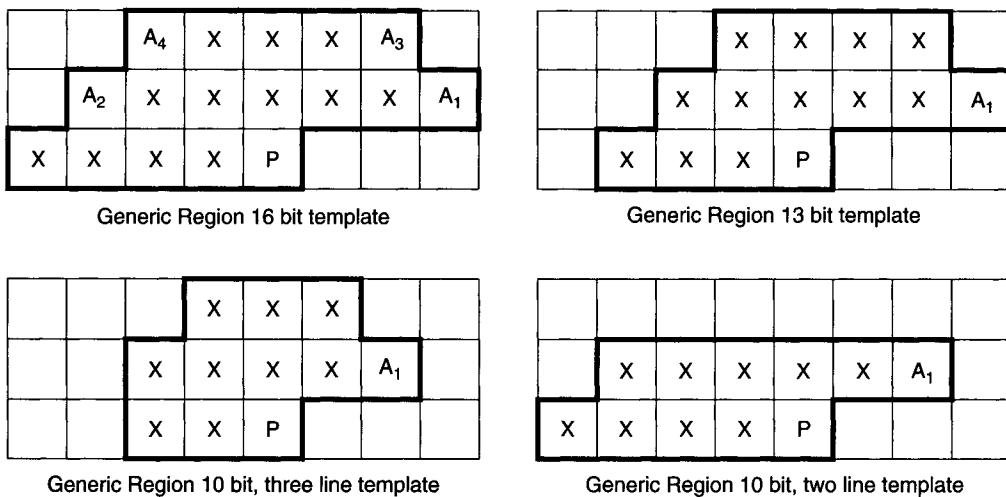
The JBIG2 standard specifies only the decoding procedures for a JBIG2 datastream. There are no normative recommendations for how a JBIG2 encoder is supposed to operate. This allows room for substantial variation in the performance of JBIG2 compatible compression products. In addition, the focus on decoding leaves room for future improvements in encoding strategies, as long as they can be cast into the rather broad and flexible description provided by the JBIG2 recommendation.

In this section the decoding procedures for the primary JBIG2 region types and dictionaries are presented with the goal of providing insight into how and why the recommendation allows substantial compression of bilevel images. The first part of this section describes the decoding required for generic regions. These regions fill or modify a buffer (i.e., a rectangular grid of bits) directly. The next part describes generic refinement region decoding. While the generic regions can be coded with either run-length MMR Huffman coding or context-based arithmetic coding, the refinements use only context-based arithmetic coding and use context bits from both the original and the refined bitmaps. The third part of this section describes the symbol dictionary decoding. Concatenated sets of similar height characters (or symbols) are encoded either directly or with an aggregate refinement based on previously defined symbols. The next part describes the decoding procedures for text regions. These regions make use of existing dictionary symbol entries and can also refine these entries in a manner consistent with the generic refinement region decoding procedures. The fifth part of this section describes the pattern dictionary decoding procedures in which the dictionary entries are concatenated and encoded as a generic region. The final part describes the halftone region decoding procedures, including the orientation of the halftone pattern grid and the pseudo-grayscale image generation and coding.

17.3.2.1 Generic Region Decoding

Generic region decoding generates a bitmap of the specified size (GBW by GBH pixels) in a straightforward manner from a received bitstream. Various control information is first decoded, including an MMR encoding flag, the size of the resulting bitmap, a template selection, and a typical prediction flag. The generic region decoding is reminiscent of both the bottom layer decoding in JBIG if adaptive arithmetic coding is used and the Group 4 facsimile standard if MMR Huffman run-length encoding is used. If the MMR flag is set to 1 then the decoding is identical to the MMR decoder described in ITU-T.6, save a few minor differences. The differences include assigning “black” pixels a value “1” and “white” pixels a value “0,” not using an EOFB symbol in cases in which the number of bytes contained in the bitmap will be known in advance and consuming an integer number of bytes in decoding the MMR data (even if bits need to be skipped at the end of a bitmap).

If the MMR flag is set to zero, then the decoding is quite similar to that used in decoding the bottom layer of a JBIG bilevel image. The size of the bitmap (two 4-byte unsigned integers) is decoded first, followed by 2 bits used for selecting the template to be used to generate the contexts for the arithmetic decoding. The four templates available for use in generic region decoding are illustrated in Fig. 17.13. There are one 16-bit, one 13-bit, and two 10-bit templates. All of the templates contain at least one adaptive template bit. The 16-bit template has four adaptive template bits. These adaptive template bits can be positioned from -128 to $+127$ bits in the horizontal (X) direction and from -128 to 0 bits in the vertical (Y) direction. Note that the two 10-bit templates are identical to those used in the bottom layer of JBIG. As with the JBIG bottom layer, bits can be

**FIGURE 17.13**

Generic region templates for context-based adaptive arithmetic coding. A_i indicates adaptive template pixels, X indicates fixed template pixels, and P indicates the pixel to be coded. Adaptive templates are shown in their default locations.

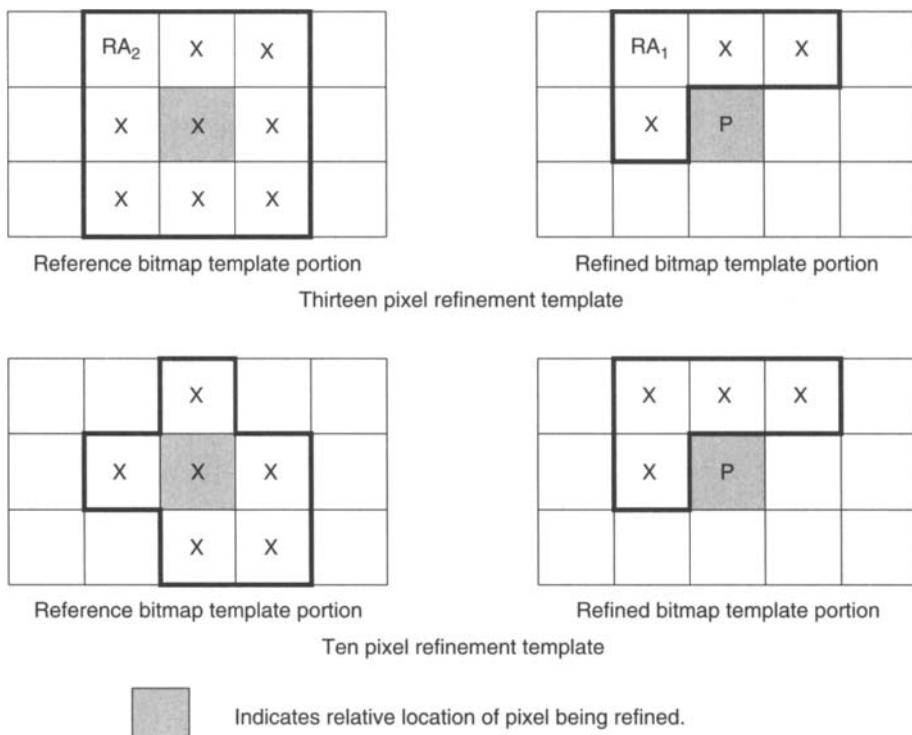
arithmetically encoded using the context provided by the templates. If template bits do not exist within the bitmap, these bits are assumed to be zero.

Typical prediction is also available in generic region decoding. If typical prediction is selected, then a pseudo-pixel is differentially encoded (via exclusive OR) as the first pixel prior to each row in the bitmap. If a given line is identical to the line immediately preceding it, then the pseudo-pixels are different from one line to the next. If the line is not identical to the preceding line, then the pseudo-pixels in the two rows are the same. In this way (as in the bottom layer decoding in JBIG) repeated lines can be represented by this single pseudo-pixel. As in JBIG, certain unlikely contexts are used to encode the pseudo-pixels.

17.3.2.2 Generic Refinement Region Decoding

The generic refinement region decoding procedure refines a reference bitmap already available to the decoder. Generic refinement uses context-based arithmetic coding to produce the refined bitmap given a reference bitmap. The set of inputs needed includes the width and height of the bitmap, a choice of templates, a reference bitmap with offsets that indicate how the two bitmaps are aligned, a typical prediction flag, and the location of the adaptive templates (if these are used). As was the case with generic region decoding, the edge convention employed assumes that any bits not located on a bitmap are zero.

Two templates are used in generic refinement decoding. Figure 17.14 illustrates both the 13- and 10-pixel templates. In Fig. 17.14, there are two grids associated with each of the two templates. One grid is from the reference bitmap, and one is from the refined bitmap. The pixels labeled X or RA_i are used in the template. These pixels are combined (in an arbitrary but consistent order) to form the context used in the arithmetic coding. Note that the pixels taken from the refined bitmap include only those that would be available to the decoder. All pixels in the reference bitmap are assumed available. The shaded pixel P indicates the location of the pixel being decoded in the refined bitmap. The shaded pixel X in the reference bitmap indicates the corresponding location of the P pixel in the reference bitmap. In the 13-bit template, the pixels RA_1 and RA_2 are the refinement adaptive templates. The restrictions on location for RA_1 are identical to those imposed on the generic region adaptive templates, while RA_2 can range from -128 to 127 in both

**FIGURE 17.14**

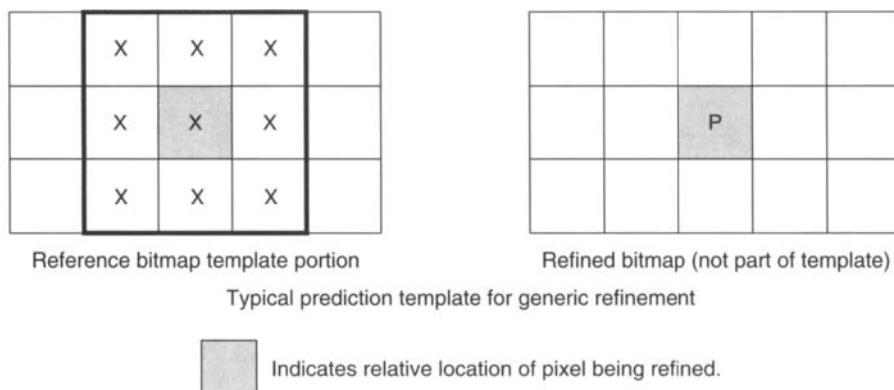
Generic refinement region templates for context-based adaptive arithmetic coding. RA_i indicates refinement adaptive template pixels, X indicates fixed template pixels, and P indicates the pixel coded in the refined bitmap. Adaptive templates are shown in their default locations.

the horizontal and vertical directions since the reference bitmap is completely available during decoding. There is no adaptive portion of the 10-bit template. The use of pixels from both the reference and refined bitmaps is similar to the use of pixels from both high and low resolutions in the differential layer encoding used in JBIG. Note, however, that the resolution of the two bitmaps is the same in generic refinement.

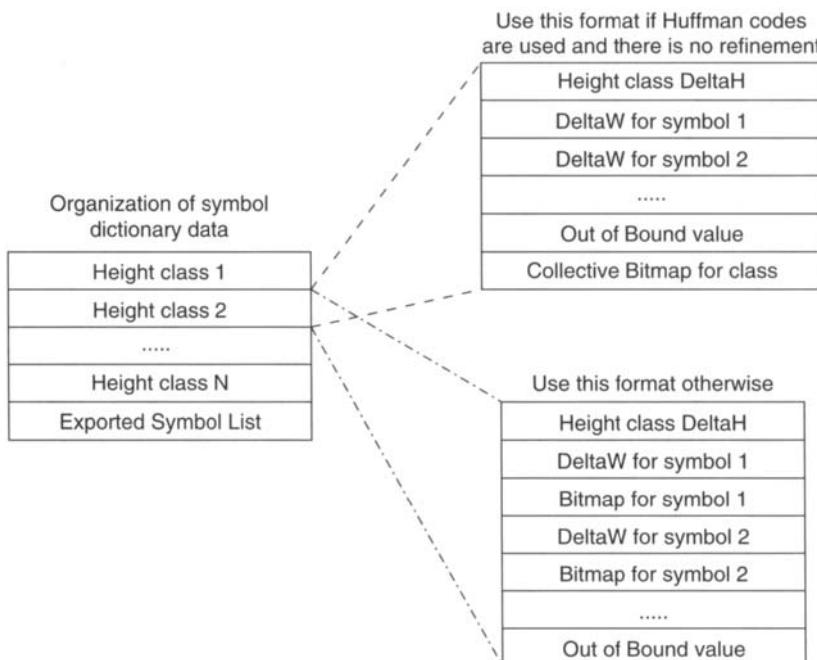
Typical prediction is allowed in generic refinement region decoding. If typical prediction is used, then a pseudo-pixel is differentially encoded at the beginning of each line to indicate whether that line is typical. If a line is deemed typical ($LTP = 1$), typical prediction allows certain pixels not to be coded. Figure 17.15 illustrates the generic refinement typical prediction template. If all of the pixels in the reference bitmap portion of the typical prediction template have the same value and $LTP = 1$, then a given pixel P in the refinement bitmap is set to that same value. If the pixels in the reference bitmap portion of the typical prediction template are not all the same, then the given pixel P in the refinement bitmap needs to be decoded.

17.3.2.3 Symbol Dictionary Decoding

The symbol dictionaries that are required to encode text segments of a document are transmitted as part of the JBIG2 datastream. These dictionary entries can then be exported, i.e., used to build up text regions or other dictionary segments, or they can merely be used as intermediate entries for building up subsequent dictionary entries, i.e., aggregate or refined dictionary entries. The data in a symbol dictionary is organized by height classes, as is illustrated in Fig. 17.16. Within each height class the data for the symbols can be encoded as one collective bitmap or individually

**FIGURE 17.15**

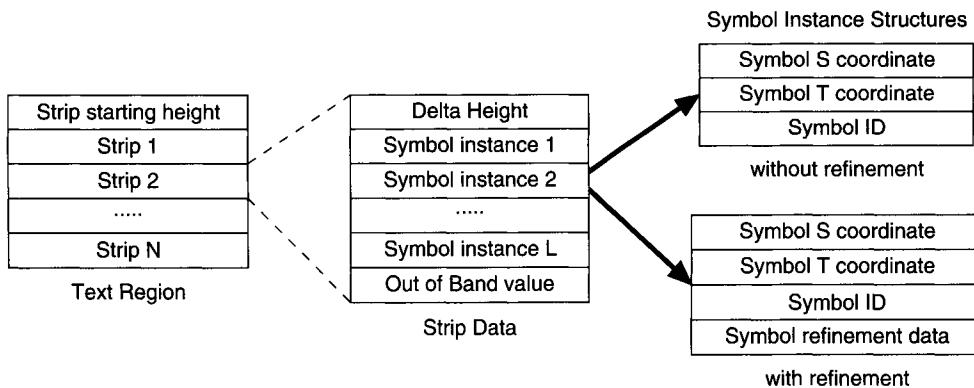
Template used for typical prediction in generic refinement. X indicates fixed template pixels and P indicates the pixel coded in the refined bitmap.

**FIGURE 17.16**

Symbol dictionary organization with height class data decoded in one of two possible ways.

as illustrated in Fig. 17.16. If Huffman coding is selected and there is to be no refinement or use of aggregate symbols, then one collective bitmap is encoded using the MMR selection of generic region encoding for the entire bitmap. The symbol width data is subsequently used to parse the bitmap into specific symbols.

If Huffman coding is not selected or if refinement or the use of aggregate symbols is used within a height class, then one of two forms of decoding is used. With no refinement or aggregation,

**FIGURE 17.17**

Text region data organization.

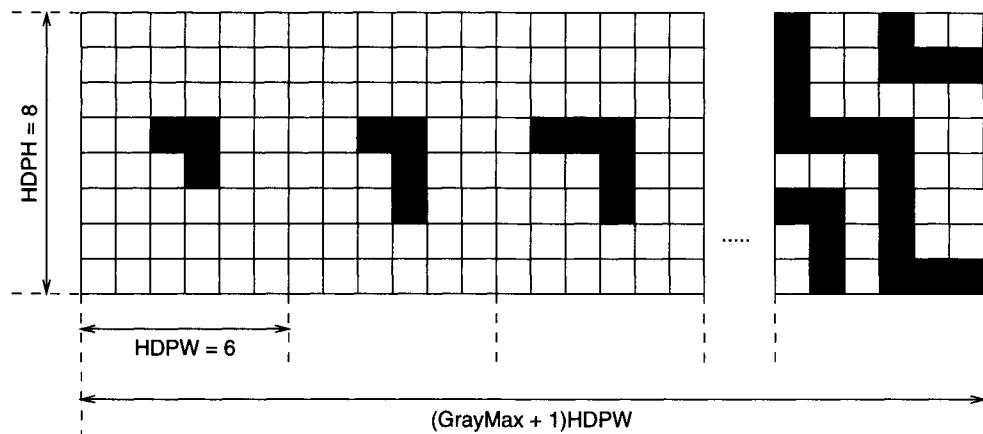
then generic region decoding is used. Otherwise, if aggregation or refinement is used, then one of two cases is decoded. If more than a single instance of previously defined symbols is used to define an aggregation, then text region decoding is used (this will be discussed in the next section). If only a single refinement is used, then a previously decoded symbol is refined using generic refinement decoding. In either case, a single bitmap is generated and placed in the dictionary. All height classes are decoded in this manner (note that *DeltaH* values can be negative). The final information decoded for the symbol dictionary is the exported symbol list. This list includes all the symbols that are available to other segments for decoding of text regions or other symbol dictionary entries.

17.3.2.4 Text Region Decoding

Text region decoding uses the indices in the symbol dictionaries to represent the bitmaps of text characters (or symbols). To do this, the S and T coordinates are decoded, along with the index of the dictionary entry for a given symbol and, if used, an optional refinement bitmap to slightly modify the dictionary entry and more closely or exactly represent the original symbol's bitmap. The S and T coordinates can be thought of as corresponding to the horizontal and vertical locations in a buffer, respectively (unless the symbols are transposed or a particular choice of reference corner determines otherwise—however, these parameters affect only the final orientation of the buffer). The data organization for text regions is indicated in Fig. 17.17. After the starting location of the first point is specified, each strip of data is decoded as a set of symbol instances until an out of band value indicates the end of that strip. Each symbol instance consists of an S and a T location update and a symbol index value. If refinement is not used, then this bitmap corresponding to this index in the dictionary is placed as specified. If refinement is used, then a refinement bitmap is decoded using generic refinement region decoding as specified previously.

17.3.2.5 Pattern Dictionary Decoding

Halftone regions are decoded using a pattern dictionary. To understand how this pattern dictionary is made available to the decoder consider the set of patterns illustrated in Fig. 17.18. Each of the entries in a pattern dictionary is positioned horizontally one after the next as shown to form a collective bitmap. This collective bitmap is decoded using generic region decoding. The number of patterns in the bitmap is given by $(GrayMax + 1)$, requiring $\lceil \log_2(GrayMax + 1) \rceil$ bits to represent each pattern, where $\lceil x \rceil$ indicates the smallest integer greater than or equal to x .



Example Collective Bitmap for Pattern Dictionary with $\text{GrayMax}+1$ entries

FIGURE 17.18

Collective bitmap for pattern dictionary entries.

GrayMax is a 4-byte unsigned integer, as are the width and height of each pattern in the dictionary (*HDPW* and *HDPH* from Fig. 17.18, respectively).

17.3.2.6 Halftone Region Decoding

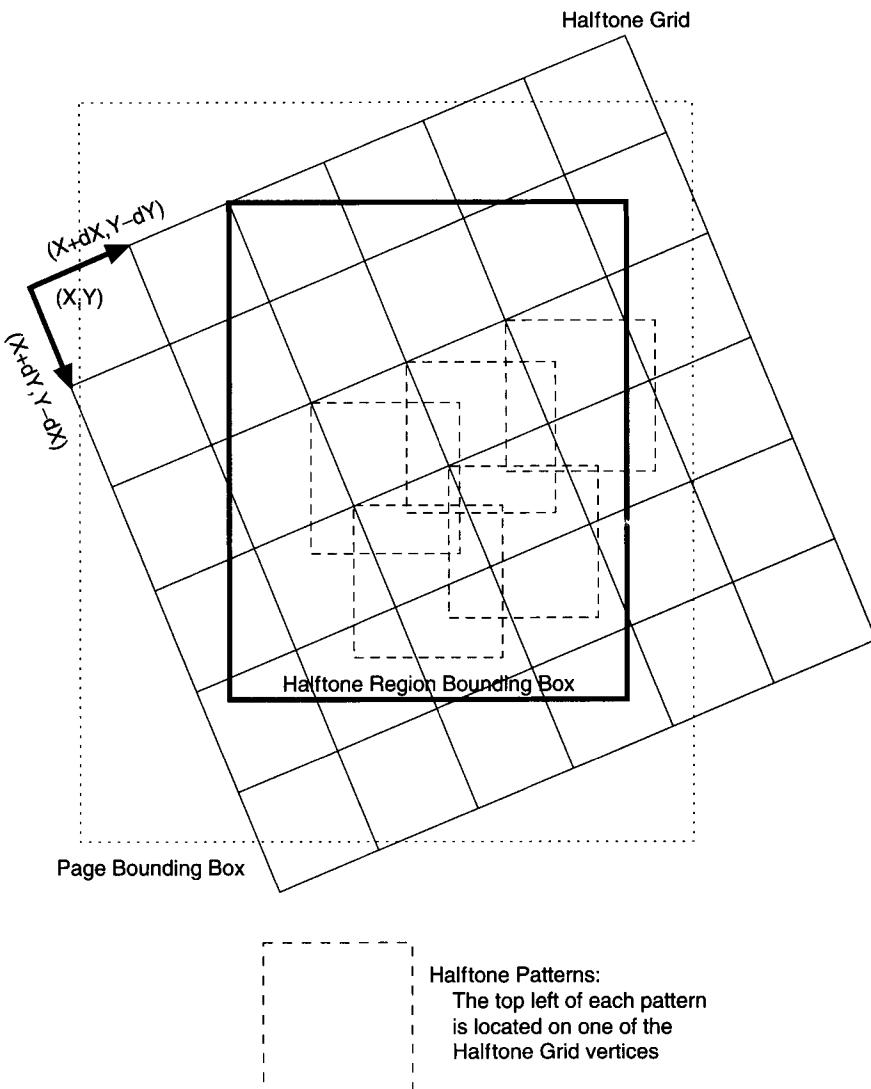
The pattern dictionary is used to decode halftone regions. Figure 17.19 illustrates the relationship between a page buffer, a halftone region, and the halftone grid. The grid is specified by parameters *X*, *Y*, *dX*, and *dY* in the figure, along with the width and height of the pseudo-grayscale image that lies on the halftone grid. The *X* and *Y* parameters are 32-bit quantities, while the *dX* and *dY* parameters are 16-bit quantities. The least significant 8 bits for all of these parameters are interpreted as a fractional component. A bitmap can be used to skip certain entries in the halftone grid which are not needed. In addition, depending on the relative angle of the halftone grid and the halftone region, there may be several grid points which are not required since they fall completely out of the halftone region or even the page buffer.

The $\lceil \log_2(\text{GrayMax} + 1) \rceil$ bitplanes of the pseudo-grayscale image are extracted (least significant bit-plane first) from a single bitmap. The dimensions of this single bitmap are the number of vertical halftone grid points by the product of the number of horizontal halftone grid points and the number of bit-planes. Generic region decoding is used to obtain this bitmap, which is organized into a pseudo-grayscale image via gray coding. Within the halftone region, these $\lceil \log_2(\text{GrayMax} + 1) \rceil$ bit values are then used as the indices into the pattern dictionary. The selected pattern from the dictionary is placed at its grid location (these are indicated in the figure by the dashed boxes). Where these patterns overlap, a set of combination operations is possible. These include logical operations such as OR, AND, and XOR, as well as a REPLACE operation in which grid locations decoded at a later time simply replace bit values decoded previously.

17.3.3 Decoding Control and Data Structures

JBIG2 datastreams are broken into segments. The primary segment types include:

- generic region segment types (immediate, intermediate, and immediate lossless)
- generic refinement segment types (immediate, intermediate, and immediate lossless)
- text region segment types (immediate, intermediate, and immediate lossless)

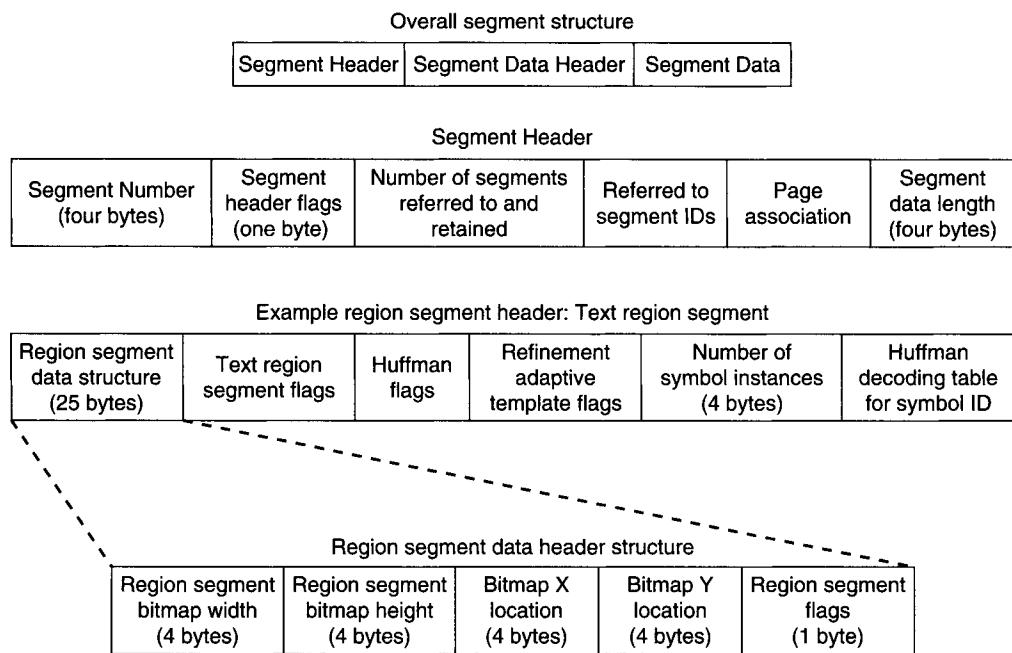
**FIGURE 17.19**

Halftone region grid for reconstruction of halftone region and representation of indices as a pseudo-grayscale image.

- halftone region segment types (immediate, intermediate, and immediate lossless)
- symbol dictionary segments
- pattern dictionary segments
- page information segments
- end of page, end of stripe, end of file, profiles, tables, and extension segments.

The final two groups of segment types control the decoding process. Most of these control segment names clearly indicate their function. As in the JBIG standard, it is possible to reduce required buffer sizes by striping page data, although this may come at the expense of reduced compression.

Within each segment there is a segment header, a segment data header, and segment data. This basic structure is indicated in Fig. 17.20. The segment header contains an ID (segment number),

**FIGURE 17.20**

JBIG2 segment and segment header structure.

1 byte of flags that include an indication of segment type, and a count of other segments which are referred to in that segment (as well as an indication of whether these other segments will need to be retained for later use—this helps the decoder free unnecessary data from memory). Other segment header information includes the segment numbers of the other segments being referred to, a page number with which the current segment is associated, and the segment data length.

An example text region segment data header is also illustrated in the figure. Region segment information, such as the width and height of the segment's bitmap and its location (X, Y) in the buffer, is included in the region segment data header. Segment flags, Huffman coding flags, refinement adaptive template flags, and the number of symbol instances to be decoded by this particular text region segment are also included. Finally, if Huffman codes are to be used for decoding the symbol instance IDs, then this table is decoded from information included in the text region data header.

17.4 SUMMARY

In this section some lossless bilevel image compression results originally presented in [2] are given to allow comparison of the relative performance of JBIG and JBIG2, as well as the T.4 and T.6 facsimile standards. Table 17.1 contains lossless bilevel image compression results for four coders and three sets of images. The four coders are G3, G4, JBIG, and JBIG2, all used for lossless compression. The three image sets include nine 200-dpi images, nine 300-dpi images, and four mask images that were generated from segmenting a page of an electronics catalog [2].

While JBIG2 is slightly better in terms of compression than JBIG, both JBIG and JBIG2 provide significant improvement relative to the G3 and G4 facsimile standards. It should be noted that if

Table 17.1 Total Number of Bits in Compressed Lossless Representation of Bilevel Image Test Sets (from [2])

Coder	Bilevel Images		
	9-200 dpi	9-300 dpi	4 mask
G3 (T.4)	647,348	1,344,030	125,814
G4 (T.6)	400,382	749,674	67,116
JBIG (T.82)	254,223	588,418	54,443
JBIG2 (T.88)	236,053	513,495	51,453

lossy compression of some parts of the bilevel images is allowed, then the JBIG2 representations may be substantially further reduced in size.

17.5 REFERENCES

1. Sayood, K., 2000. *Introduction to Data Compression*, 2nd ed., Morgan-Kaufmann, San Mateo, CA.
2. Malvar, H. S., 2001. Fast adaptive encoder for bi-level images. *Proceedings of the Data Compression Conference*, March 27–29, 2001, pp. 253–262.
3. ITU-T Recommendation T.82, 1993. Information Technology—Progressive Bi-Level Image Compression, JBIG Recommendation.
4. ITU-T Recommendation T.88, 2000. Information Technology—Lossy/Lossless Coding Bi-Level Images, JBIG2 Recommendation.

This Page Intentionally Left Blank

JPEG2000: Highly Scalable Image Compression

ALI BILGIN
MICHAEL W. MARCELLIN

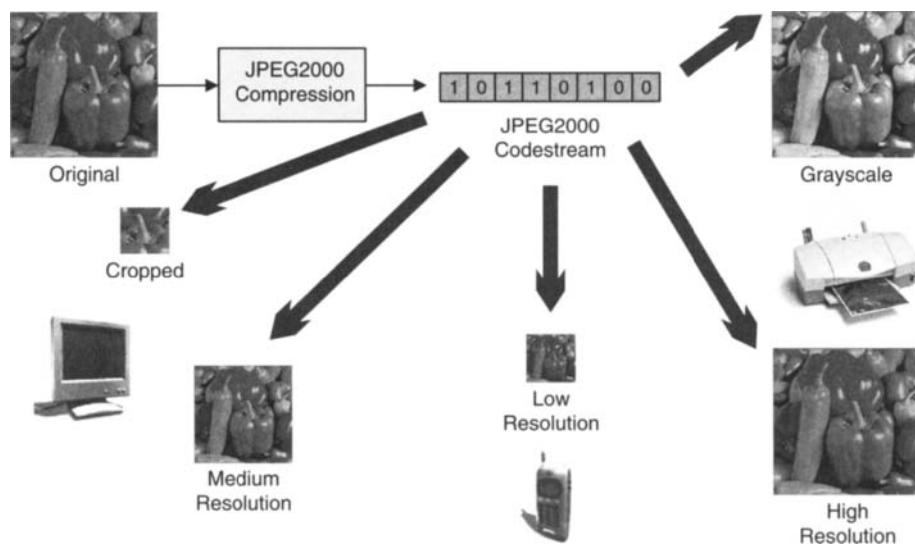
18.1 INTRODUCTION

JPEG2000 is the latest international standard for image compression [1, 2]. In addition to providing state-of-the-art compression performance, it offers a number of functionalities that address the requirements of emerging imaging applications. Previous image compression systems and/or standards have been used primarily as input–output filters within applications. That is, as an image was written to (or read from) a disk it was compressed or decompressed, largely as a storage function. Additionally, all decisions as to image quality and/or compression ratio were made at compression time. At decompression time, only the (single) image quality, size, resolution, and spatial extent envisioned by the compressor were available to the decompressor.

For example, with JPEG baseline (sequential mode), an image is compressed using a particular quantization table. This essentially determines the quality that will be achieved at decompression time. Lower (or higher) quality decompressions are not available to the decompressor. Similarly, if JPEG lossless or JPEG-LS [3] is employed for compression, only lossless decompression is available, and high compression ratios are impossible.

Notable exceptions to these rigid structures exist. In fact, the original JPEG standard has four “modes” of operation: sequential, progressive, hierarchical, and lossless. However, these four modes rely on distinctly different technologies and result in distinctly different codestreams. While certain interactions (say, between progressive and hierarchical) are allowed according to the standard document, this has never been exploited (to the best of the authors’ knowledge) in a commercial system.

JPEG2000 creates a framework where the image compression system acts more like an image processing system than a simple input–output storage filter. The decision on several key compression parameters such as quality or resolution can be delayed until after the creation of the codestream and several different image products can be extracted from a single codestream. Some of this functionality is illustrated in Fig. 18.1 (see also color insert).

**FIGURE 18.1**

Some of the functionality offered by JPEG2000. (See also color insert.)

The JPEG2000 standard will be published in six parts. Part I will describe the minimal decoder and the codestream syntax that must be followed for compliance with the standard. Part II will consist of “value-added” technologies that improve the performance and Part III of the standard will define “Motion JPEG2000,” which includes extensions for image sequences. Part IV will provide information on compliance, Part V will contain reference software, and Part VI will describe an extended file format for document imaging.

18.2 JPEG2000 FEATURES

JPEG2000 brings a new paradigm to image compression standards [1, 2]. The benefits of all four JPEG modes are tightly integrated in JPEG2000. The compressor decides maximum image quality (up to and including lossless). Also chosen at compression time is the maximum resolution (or size). Any image quality or size can be decompressed from the resulting codestream, up to and including the maxima chosen at encode time.

For example, suppose an image is compressed losslessly at full size. Suppose further that the resulting file is of size B_0 (bytes). It is then possible to extract B_1 bytes from the file ($B_1 < B_0$) and decompress those B_1 bytes to obtain a lossy decompressed image. Similarly, it is possible to extract B_2 bytes from the file and decompress to obtain a reduced resolution image. In addition to the quality scalability and resolution scalability, JPEG2000 codestreams support spatial random access. There are several mechanisms to retrieve and decompress data from the codestream corresponding to arbitrary spatial regions of an image. The different mechanisms yield different granularity of access, at varying levels of difficulty.

This random access extends to color components as well. Specifically, the black and white (grayscale) component can be extracted from a color image. As above, this can be done region by region with varying qualities and resolutions.

It is important to note that in every case discussed above, it is possible to locate, extract, and decode only the bytes required to decode the desired image product. It is *not* necessary to decode

the entire codestream and/or image. In many cases, the bytes extracted and decoded are identical to those obtained if only the desired image products were compressed in the first place.

18.2.1 Compressed Domain Image Processing/Editing

Any of the image products discussed above can be extracted to create a new JPEG2000 compliant codestream without a decompress/recompress cycle. In addition to the elegance and computational savings, compression noise “build-up” that occurs in most compression schemes when repetitive compress/decompress cycles are utilized can be avoided.

In addition to reduced quality and reduced resolutions, compressed domain image cropping is possible. Cropping in the compressed domain is accomplished by accessing the compressed data associated with a given spatial region (using random codestream access, as discussed above) and rewriting it as a compliant codestream. Some special processing is required around the cropped image borders.

18.2.2 Progression

Many types of progressive transmission are supported by JPEG2000. As mentioned previously, progressive transmission is highly desirable when receiving imagery over slow communication links. As more data are received, the rendition of the displayed imagery improves in some fashion. JPEG2000 supports progression in four dimensions: quality, resolution, spatial location, and component.

The first dimension of progressivity in JPEG2000 is quality. As more data are received, image quality is improved. It should be noted that the image quality improves remarkably quickly with JPEG2000. An image is typically recognizable after only about 0.05 bits/pixel have been received. For a 320×240 pixel image, this corresponds to only 480 bytes of received data. With only 0.25 bits/pixel (2,400 bytes) received, most major compression artifacts disappear. To achieve quality corresponding to no visual distortion, between 0.75 and 1.0 bits/pixel are usually required. Demanding applications may sometimes require up to 2.0 bits/pixel or even truly lossless decompression (e.g., medical applications). We remark here again that all qualities up to and including lossless (equivalently, all bit-rates, or all compression ratios) are contained within a single compressed codestream. Improving quality is a simple matter of decoding more bits.

The second dimension of progressivity in JPEG2000 is resolution. In this type of progression, the first few bytes are used to form a small “thumbnail” of the image. As more bytes are received, the resolution (or size) of the image increases by factors of 2 (on each side). Eventually, the full-size image is obtained.

The third dimension of progressivity in JPEG2000 is spatial location. With this type of progression, imagery is received in a “stripe-” or “scan-based” fashion, from top to bottom. This type of progression is particularly useful for certain types of low-memory printers and scanners.

The fourth and final dimension of progressivity is classified by component. JPEG2000 supports images with up to 16,384 components. Most images with more than 4 components are from scientific instruments (e.g., LANDSAT). More typically, images are 1 component (grayscale), 3 components (e.g., RGB, YUV), or 4 components (CMYK). Overlay components containing text or graphics are also common. Component progression controls the order in which the data corresponding to different components are decoded. With progressivity by component, the grayscale version of an image might first be decoded, followed by color information, followed by overlaid annotations, text, etc.

The four dimensions of progressivity are very powerful and can be changed (nearly at will) throughout the codestream. Since only the data required by the viewer need to be transmitted, the “effective compression ratio” experienced by the client can be many times greater than the actual compression ratio as measured from the file size at the server.

18.3 THE JPEG2000 ALGORITHM

In this section, we provide a high-level description of the JPEG2000 algorithm. Although the standard specifies only the decoder and codestream syntax, this review focuses on the description of a representative encoder, since this enables a more readable explanation of the algorithm. We note that we will discuss the algorithm as it applies to Part I of the standard. As mentioned earlier, Part I describes the minimal decoder required for JPEG2000, which should be used to provide maximum interchange. Value-added technologies that are not required of all implementations are described in Part II.

18.3.1 Tiles and Component Transforms

In JPEG2000, an image is defined as a collection of two-dimensional rectangular arrays of samples. Each of these arrays is called an image component. Typical examples include RGB images that have three components and CMYK images that have four components. Components need not have the same number of samples. The image resides on a high-resolution grid. This grid, usually referred to as “the canvas,” is the reference for all geometric structures in JPEG2000. The main purpose of the canvas is to define a consistent set of region mapping rules to be used for manipulation of the geometric structures of JPEG2000.

The first step in JPEG2000 is to divide the image into non-overlapping rectangular tiles. The array of samples from one component that fall within a tile is called a *tile-component*. The primary benefits of tiles are that they provide a simple vehicle for limiting implementation memory requirements and for spatial random access. They can also be useful for segmenting compound imagery, as the coding parameters can be changed from tile to tile. As the tile grid is rectangular and regular, options for segmentation are rather restricted.

The primary disadvantage of tiles is blocking artifacts. Since each tile is compressed independently of all other tiles, visible artifacts can occur at tile boundaries. For high bit-rates and large tile sizes, these artifacts are generally invisible. The tile size can always be made so large as to encompass the entire image, effectively eliminating the presence of tiles.

When multiple component images are being encoded, one of two optional component transforms can be applied to the first three components.¹ These transforms decorrelate the components and increase the compression efficiency.

The first component transform is the irreversible color transform (ICT) and is used only in conjunction with the irreversible wavelet transform discussed in the following section. The second transform is called the reversible color transform (RCT) and is used only in conjunction with the reversible wavelet transform. Although both transforms are invertible in the mathematical sense, the RCT maps integer color components to integer transformed color components and is perfectly invertible using only finite (low) precision arithmetic. Conversely, the ICT employs floating-point arithmetic and, in general, requires infinite precision arithmetic to guarantee perfect inversion. The RCT can be viewed as an approximate version of ICT.

¹ Part II of the standard enables more general component transforms.

Let $X_R[\mathbf{n}]$, $X_G[\mathbf{n}]$, and $X_B[\mathbf{n}]$, $\mathbf{n} = [n_1, n_2]$, denote the samples for the first three components. These components would typically be the R, G, and B components of a color image. However, this is not a requirement and the color transform can be utilized regardless of the color interpretation of the first three components. The ICT is then defined as

$$\begin{aligned} X_Y[\mathbf{n}] &\triangleq \alpha_R X_R[\mathbf{n}] + \alpha_G X_G[\mathbf{n}] + \alpha_B X_B[\mathbf{n}] \\ X_{C_b}[\mathbf{n}] &\triangleq \frac{0.5}{1 - \alpha_B} (X_B[\mathbf{n}] - X_Y[\mathbf{n}]) \\ X_{C_r}[\mathbf{n}] &\triangleq \frac{0.5}{1 - \alpha_R} (X_R[\mathbf{n}] - X_Y[\mathbf{n}]), \end{aligned} \quad (18.1)$$

where

$$\alpha_R \triangleq 0.299, \quad \alpha_G \triangleq 0.587, \quad \alpha_B \triangleq 0.114.$$

The RCT is defined as

$$\begin{aligned} X_{Y'}[\mathbf{n}] &\triangleq \left\lfloor \frac{X_R[\mathbf{n}] + 2X_G[\mathbf{n}] + X_B[\mathbf{n}]}{4} \right\rfloor \\ X_{D_b}[\mathbf{n}] &\triangleq X_B[\mathbf{n}] - X_{G'}[\mathbf{n}] \\ X_{D_r}[\mathbf{n}] &\triangleq X_R[\mathbf{n}] - X_{G'}[\mathbf{n}]. \end{aligned} \quad (18.2)$$

Notice that the output quantities of this transform are all integers, since the inputs to the transform are integers. The RCT can be exactly inverted using

$$\begin{aligned} X_G[\mathbf{n}] &= X_{Y'}[\mathbf{n}] - \left\lfloor \frac{X_{D_b}[\mathbf{n}] + X_{D_r}[\mathbf{n}]}{4} \right\rfloor \\ X_B[\mathbf{n}] &= X_{D_b}[\mathbf{n}] + X_G[\mathbf{n}] \\ X_R[\mathbf{n}] &= X_{D_r}[\mathbf{n}] + X_G[\mathbf{n}]. \end{aligned} \quad (18.3)$$

18.3.2 The Wavelet Transform

In this section, we provide a brief overview of the wavelet transforms used in JPEG2000. Our primary objective is to illustrate the implementation of the wavelet transform in Part I of the JPEG2000 standard. For a more detailed review of the subband and wavelet transforms, the interested reader is referred to several excellent texts on these subjects [4–7].

JPEG2000 (Part I) utilizes two-channel subband transforms with linear phase filters. Figure 18.2 shows a two-channel filter bank. In the figure, h_0 and h_1 are analysis filters, and g_0 and g_1 are

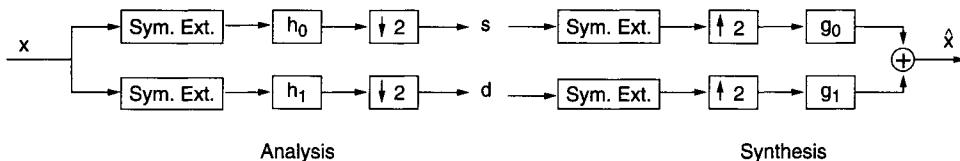
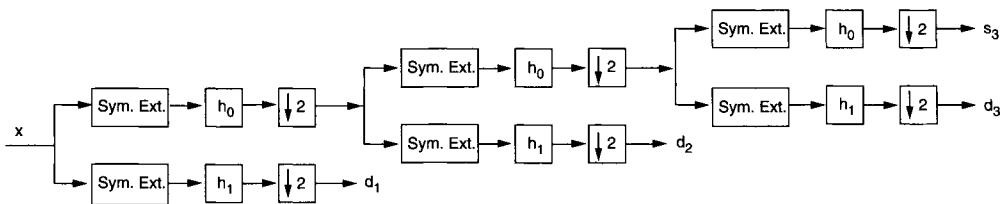


FIGURE 18.2

Wavelet analysis and synthesis.

**FIGURE 18.3**

Dyadic wavelet analysis.

synthesis filters. In the analysis step, the discrete-time input signal $x[n]$ is symmetrically extended to form $\tilde{x}[n]$, filtered using h_0 and h_1 , and down-sampled to generate the low-pass band $s[n]$ and the high-pass band $d[n]$. The total number of samples in $s[n]$ and $d[n]$ is equal to the number of samples in $x[n]$, so the transform is non-expansive. At the synthesis stage, $s[n]$ and $d[n]$ are symmetrically extended, up-sampled, and filtered with g_0 and g_1 , respectively. The sum of the filter outputs results in the reconstructed signal $\hat{x}[n]$. It is possible to further decompose $s[n]$ and $d[n]$. In a dyadic decomposition, the lowest frequency band is decomposed in a recursive fashion. We refer to the number of these recursive steps as dyadic levels. Figure 18.3 illustrates a three-level dyadic decomposition.

The wavelet transform can be extended to multidimensions using separable filters. Each dimension is filtered and down-sampled separately. Thus, the two-dimensional wavelet transform is achieved by applying the one-dimensional subband transform first to the columns and then the rows of the image. The four subbands produced by this operation are labeled according to the filters used in the horizontal and vertical directions, respectively. Thus, the labels used to identify the four subbands resulting from a two-dimensional wavelet transform are LL, HL, LH, and HH. Subsequent levels can be obtained by applying the transform to the the lowest frequency (LL) band in a recursive fashion. Figure 18.4 illustrates a three-level two-dimensional wavelet transform.

JPEG2000 (Part I) supports two choices for the wavelet filters. The so-called (9,7) wavelet transform [8] utilizes filters with floating-point impulse responses of lengths 9 and 7. The low- and high-pass analysis filters for this transform are given approximately by

$$\begin{aligned} h_0(z) = & 0.602949018236 + 0.266864118443(z^1 + z^{-1}) \\ & - 0.078223266529(z^2 + z^{-2}) - 0.016864118443(z^3 + z^{-3}) \\ & + 0.026748757411(z^4 + z^{-4}); \end{aligned}$$

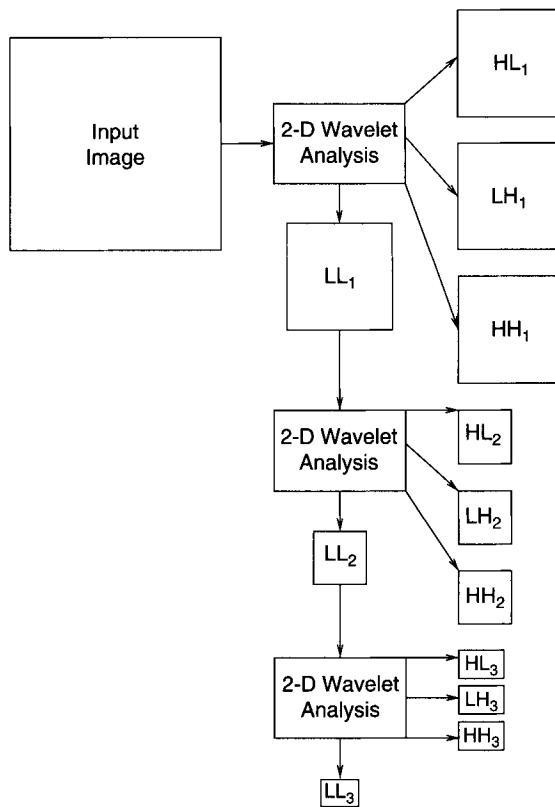
$$\begin{aligned} h_1(z) = & 0.557543526229 - 0.295635881557(z^1 + z^{-1}) \\ & - 0.028771763114(z^2 + z^{-2}) + 0.045635881557(z^3 + z^{-3}), \end{aligned}$$

and the low- and high-pass synthesis filters can be deduced from the analysis filters using

$$\begin{aligned} g_0[n] &= \alpha^{-1}(-1)^n h_1[n] \\ g_1[n] &= \alpha^{-1}(-1)^n h_0[n], \end{aligned}$$

where α is a gain factor and equals $\frac{1}{2}$ for the analysis filters given above. This transform is known as the irreversible transform and is useful for high-performance lossy compression.

The so-called (5, 3) wavelet transform [9] is implemented using integer arithmetic. Careful rounding of both intermediate results and the final wavelet coefficients is performed during filtering. The resulting transform is reversible, enabling lossless (in addition to lossy) compression.

**FIGURE 18.4**

Two-dimensional wavelet analysis.

For the (5, 3) wavelet transform, the analysis operation is defined by

$$\begin{aligned}\tilde{d}[n] &= \tilde{x}[2n+1] + \left[\frac{1}{2} - \frac{1}{2}\tilde{x}[2n] - \frac{1}{2}\tilde{x}[2n+2] \right] \\ \tilde{s}[n] &= \tilde{x}[2n] + \left[\frac{1}{2} + \frac{1}{4}\tilde{d}[n-1] + \frac{1}{4}\tilde{d}[n] \right],\end{aligned}$$

and the synthesis operation is defined by

$$\begin{aligned}\tilde{x}[2n] &= \tilde{s}[n] - \left[\frac{1}{2} + \frac{1}{4}\tilde{d}[n-1] + \frac{1}{4}\tilde{d}[n] \right] \\ \tilde{x}[2n+1] &= \tilde{d}[n] - \left[\frac{1}{2} - \frac{1}{2}\tilde{x}[2n] - \frac{1}{2}\tilde{x}[2n+2] \right],\end{aligned}$$

where $\tilde{s}[n]$ and $\tilde{d}[n]$ are symmetrically extended versions of $s[n]$ and $d[n]$, respectively.

For a given (lossy) compression ratio, the image quality obtained with the (9, 7) transform is generally superior to that obtained with the (5, 3) transform. However, the performance of the (5, 3) transform is still quite good.

Each resolution of a tile-component is partitioned into *precincts*. Precincts behave much like tiles, but in the wavelet domain. The precinct size can be chosen independently by resolution; however, each side of a precinct must be a power of 2 in size. Figure 18.5 shows a precinct partition for a single resolution.

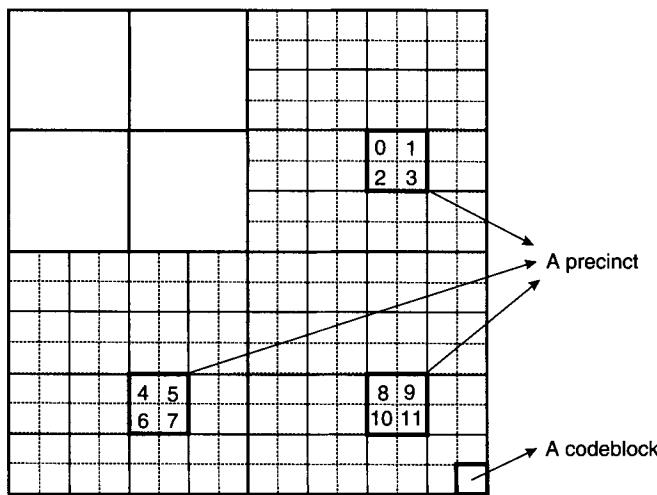


FIGURE 18.5
Partitioning of wavelet subbands.

Precincts are another ingredient to low-memory implementations in the absence of tiles. Compressed data from a precinct are grouped together to form a *packet*. Before a packet header can be created, all compressed data from the corresponding precinct must be available. Thus, only the compressed data for a precinct must be buffered, rather than the data of an entire tile (or image in the absence of tiles).

In addition to the memory benefits of precincts, they provide a method of spatial random access. The granularity of this method is finer than that for tiles, but coarser than that for *codeblocks* (see next paragraph). Also, since precincts are formed in the wavelet transform domain, there is no “break” in the transform at precinct boundaries (as there is at tile boundaries). Thus, precincts do not cause block (tile) artifacts.

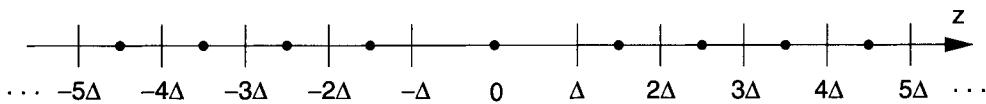
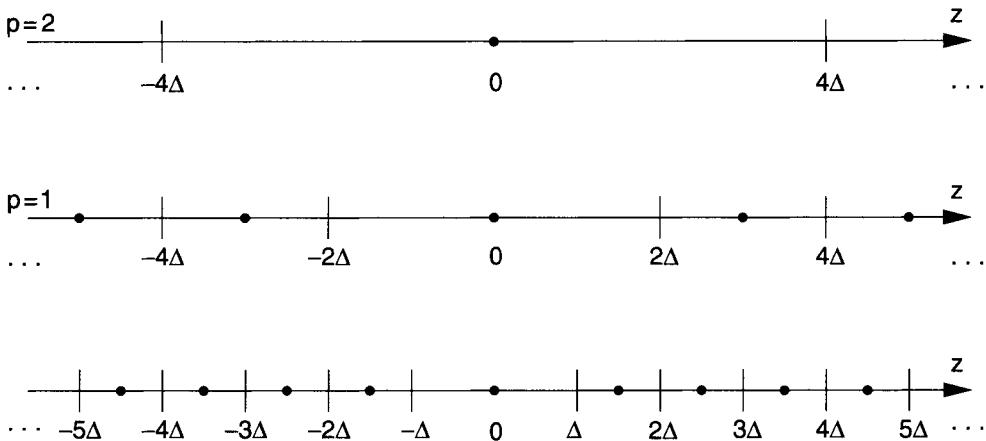
Codeblocks form the smallest geometric structure of JPEG2000. Initial quantization and bit-plane coding are performed on codeblocks. Codeblocks are formed by partitioning subbands. Since the precinct size (resolution dependent) and codeblock size (resolution independent) are both powers of 2, the two partitions are forced to “line up.” Thus, it is reasonable to view the codeblocks as partitions of the precincts (rather than of the subbands). If the codeblock size exceeds the precinct size in any subband, the codeblocks are forced to obey precinct boundaries. Effectively then, the codeblock width or height (or both) is reduced to that of the precinct. The partitioning of wavelet subbands into codeblocks is illustrated in Fig. 18.5 as well.

As discussed earlier, codeblocks help enable low-memory implementation by limiting the amount of (uncompressed) wavelet data that must be buffered before it can be quantized and compressed. This is in contrast to precincts, which limit the amount of compressed data that must be buffered before packets can be formed. Another benefit of codeblocks is that they provide fine-grain random access to spatial regions.

18.3.3 Quantization

Quantization is a process by which wavelet coefficients are reduced in precision to increase compression. For a given wavelet coefficient z and quantizer step size Δ , a signed integer q given by

$$q = Q(z) = \text{sign}(z) \left\lfloor \frac{|z|}{\Delta} \right\rfloor \quad (18.4)$$

**FIGURE 18.6**Dead-zone uniform scalar quantizer with step size Δ .**FIGURE 18.7**

Embedded dead-zone uniform scalar quantizer.

is used to indicate in which interval z lies. This index is coded using techniques described later in this section. Given q , the decoder must estimate the value of z as

$$\hat{z} = \overline{Q^{-1}}(q) = \begin{cases} 0 & q = 0 \\ \text{sign}(q)(|q| + \delta)\Delta & q \neq 0, \end{cases}$$

where δ is a user-selectable parameter (typically $\delta = 1/2$). This particular type of quantization is known as dead-zone uniform scalar quantization and is illustrated in Fig. 18.6.

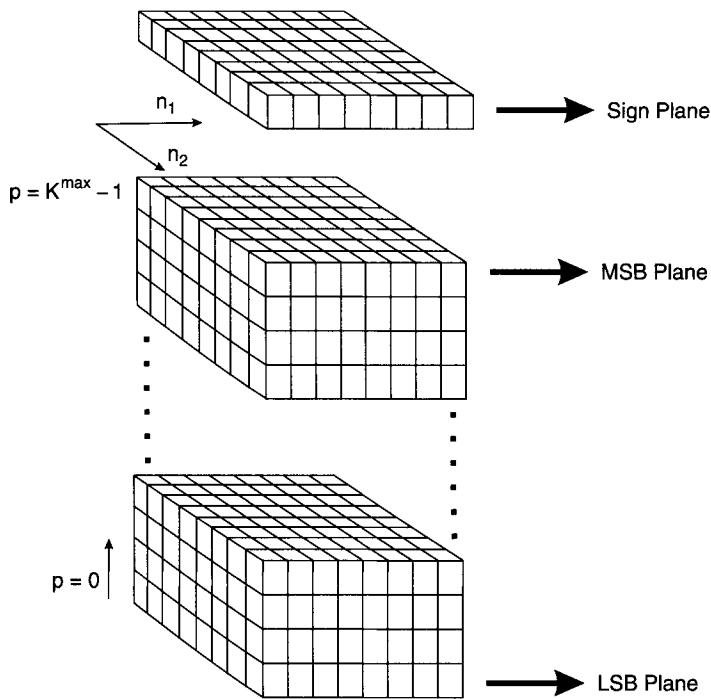
The dead-zone uniform scalar quantization is key to the quality progression feature of JPEG2000. The reason for this is that such quantizers are “embedded.” That is, the quantization index of every quantization (with step size Δ) has embedded within it the index of every quantization with step size $2^p\Delta$, $p = 0, 1, 2, \dots$. Thus, if z is quantized with step size Δ to get q , and the p Least Significant Bits (LSBs) of q are missing (not decoded yet), then the appropriate dequantizer is

$$\hat{z} = \overline{Q^{-1}}(q^{(p)}) = \begin{cases} 0 & q^{(p)} = 0 \\ \text{sign}(q^{(p)})(|q^{(p)}| + \delta)2^p\Delta & q^{(p)} \neq 0. \end{cases} \quad (18.5)$$

In this case, \hat{z} is identical to what it would have been if the step size had been $2^p\Delta$ in the first place. Figure 18.7 illustrates this embedding for $p = 1$ and $p = 2$.

For irreversible wavelets, a different step size can be chosen for each subband. These step sizes are substantially equivalent to Q -table values from JPEG and can be chosen to meet differing needs. For reversible wavelets, a step size of $\Delta = 1$ is used. This results in no quantization at all unless one or more LSBs of the (integer) wavelet coefficients are omitted. In this case, the equivalent induced step size is $2^p\Delta = 2^p$.

The inverse quantizer formula given in Eq. (18.5) has a problem when followed by a reversible transform. There is no guarantee that the estimate \hat{z} is an integer. To avoid this problem, JPEG2000

**FIGURE 18.8**

Bit-plane representation of a codeblock.

modifies the inverse quantization definition in this special case to

$$\hat{z} = \overline{Q_p^{-1}}(q^{(p)}) = \begin{cases} 0 & q^{(p)} = 0 \\ \text{sign}(q^{(p)}) \lfloor (|q^{(p)}| + \delta) 2^p \Delta \rfloor & q^{(p)} \neq 0. \end{cases} \quad (18.6)$$

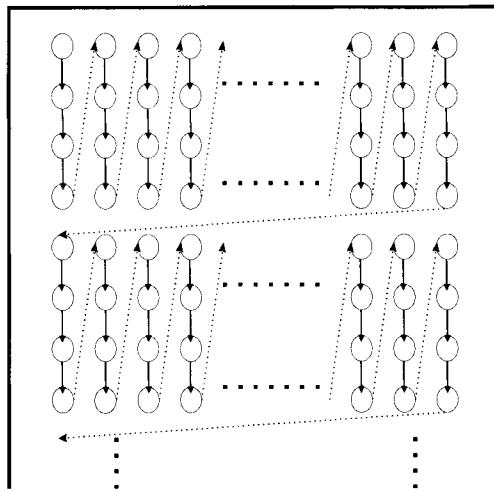
For a detailed review of quantization in JPEG2000, the interested reader is referred to [2, 10].

18.3.4 Bit-Plane Coding

Entropy coding is performed independently on each codeblock. This coding is carried out as context-dependent, binary, arithmetic coding of bit-planes. The arithmetic coder employed is the MQ coder as specified in the JBIG-2 standard.

Consider a quantized codeblock to be an array of integers in sign-magnitude representation. Let $q[\mathbf{n}]$ denote the quantization index at location $\mathbf{n} = [n_1, n_2]$ of the codeblock. Let $\chi[\mathbf{n}] \triangleq \text{sign}(q[\mathbf{n}])$ and $v[\mathbf{n}] \triangleq |q[\mathbf{n}]|$ denote the sign and magnitude arrays, respectively.² Then, consider a sequence of binary arrays with 1 bit from each coefficient. These binary arrays are referred to as *bit-planes*. One such array can store the signs of each coefficient. Let the number of magnitude bit-planes for the current subband be denoted by K^{\max} . The first magnitude bit-plane contains the most significant bit (MSB) of all the magnitudes. The second bit-plane contains the next MSB of all the magnitudes, continuing in this fashion until the final bit-plane, which consists of the LSBs of all the magnitudes. This representation is illustrated in Fig. 18.8.

² It should be noted that $\chi[\mathbf{n}]$ is indeterminate when $q[\mathbf{n}] = 0$. However, this case will not cause any problems, since the sign need not be coded when $q[\mathbf{n}] = 0$.

**FIGURE 18.9**

Scan pattern for bit-plane coding.

Let

$$v^{(p)}[\mathbf{n}] \triangleq \left\lfloor \frac{v[\mathbf{n}]}{2^p} \right\rfloor$$

denote the value generated by dropping p LSBs from $v[\mathbf{n}]$. The “significance” of a sample is then defined by

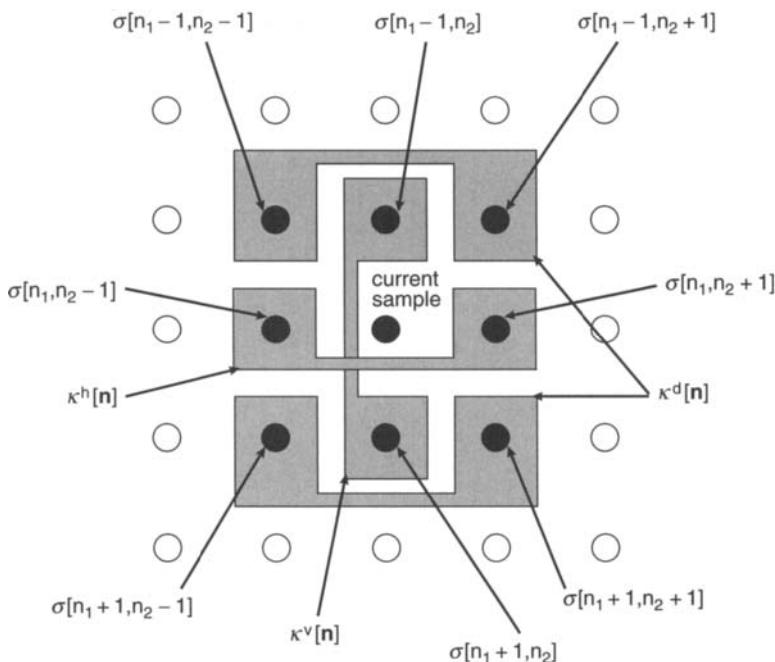
$$\sigma^{(p)}[\mathbf{n}] \triangleq \begin{cases} 1 & v^{(p)}[\mathbf{n}] > 0 \\ 0 & v^{(p)}[\mathbf{n}] = 0. \end{cases} \quad (18.7)$$

Thus, a sample at location \mathbf{n} is said to be significant with respect to bit-plane p , if $\sigma^{(p)}[\mathbf{n}] = 1$. Otherwise, the sample is considered to be insignificant. For a given codeblock in the subband, the encoder first determines the number of bit-planes, $K \leq K^{\max}$, that are required to represent the samples in the codeblock. In other words, K is selected as the smallest integer such that $v[\mathbf{n}] < 2^K$ for all \mathbf{n} . The number of bit-planes (starting from the MSB) which are identically zero, $K^{\text{msbs}} = K^{\max} - K$, is signaled to the decoder as side information.³ Then, starting from the first bit-plane having at least a single 1, each bit-plane is encoded in three passes (referred to as coding passes).

The scan pattern followed for the coding of bit-planes, within each codeblock (in all subbands), is shown in Fig. 18.9. This scan pattern is followed in each of the three coding passes. The decision as to which pass a given bit is coded in is made based on the significance of that bit's location and the significance of neighboring locations. All bit-plane coding is done using context-dependent binary arithmetic coding with the exception that run-coding is sometimes employed in the third pass. Let us define $\sigma[\mathbf{n}]$ as the “significance state” of a sample during the coding process. $\sigma[\mathbf{n}]$ is set to zero for all samples at the start of the coding process and it is reset to 1 as soon as the first non-zero magnitude of the sample is coded.

The first pass in a new bit-plane is called the significance propagation pass. A bit is coded in this pass if its location is not significant, but at least one of its eight-connected neighbors is significant.

³ It should also be noted that the K^{\max} of each subband is also available at the decoder.

**FIGURE 18.10**

Significance context modeling.

In other words, a bit at location \mathbf{n} is coded in a significance propagation pass, if $\sigma[\mathbf{n}] = 0$ and

$$\sum_{k_1=-1}^1 \sum_{k_2=-1}^1 \sigma[n_1 + k_1, n_2 + k_2] \geq 1.$$

If the bit that is coded is 1, then the sign bit of the current sample is coded and $\sigma[\mathbf{n}]$ is set to 1 immediately.

The arithmetic coding in JPEG2000 employs different context models depending on the coding pass and the subband type. For significance coding, nine different contexts are used. The context label $\kappa^{sig}[\mathbf{n}]$ is dependent on the significance states of the eight-connected neighbors of the current bit. This is illustrated in Fig. 18.10. Using these neighboring significance states, $\kappa^{sig}[\mathbf{n}]$ is formed from three quantities:

$$\begin{aligned}\kappa^h[\mathbf{n}] &= \sigma[n_1, n_2 - 1] + \sigma[n_1, n_2 + 1] \\ \kappa^v[\mathbf{n}] &= \sigma[n_1 - 1, n_2] + \sigma[n_1 + 1, n_2] \\ \kappa^d[\mathbf{n}] &= \sum_{k_1=\pm 1} \sum_{k_2=\pm 1} \sigma[n_1 + k_1, n_2 + k_2].\end{aligned}$$

Table 18.1 shows how $\kappa^{sig}[\mathbf{n}]$ is generated given $\kappa^h[\mathbf{n}]$, $\kappa^v[\mathbf{n}]$, and $\kappa^d[\mathbf{n}]$. $\kappa^{sig}[\mathbf{n}]$ then determines the probability estimate that will be used in arithmetic coding.

As stated earlier, the sign bit of a sample is coded immediately after its first non-zero bit. Similar to significance coding, sign coding also employs context modeling techniques, since it has been observed that the signs of neighboring samples exhibit statistical redundancy. JPEG2000 uses five contexts for sign coding. The context label $\kappa^{sign}[\mathbf{n}]$ is selected depending on the four-connected neighbors of the current samples. The neighborhood information is incorporated

Table 18.1 Context Labels for Significance Coding

$\kappa^{sig}[\mathbf{n}]$	LL and LH Codeblocks			HL Codeblocks			HH Codeblocks	
	$\kappa^h[\mathbf{n}]$	$\kappa^v[\mathbf{n}]$	$\kappa^d[\mathbf{n}]$	$\kappa^h[\mathbf{n}]$	$\kappa^v[\mathbf{n}]$	$\kappa^d[\mathbf{n}]$	$\kappa^d[\mathbf{n}]$	$\kappa^v[\mathbf{n}] + \kappa^d[\mathbf{n}]$
8	2	—	—	—	2	—	≥ 3	—
7	1	≥ 1	—	≥ 1	1	—	2	≥ 1
6	1	0	≥ 1	0	1	≥ 1	2	0
5	1	0	0	0	1	0	1	≥ 2
4	0	2	—	2	0	—	1	1
3	0	1	—	1	0	—	1	0
2	0	0	≥ 2	0	0	≥ 2	0	≥ 2
1	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0	0

Note: A dash denotes “don’t care.”

Table 18.2 Context Labels for Sign Coding

$\bar{\chi}^h[\mathbf{n}]$	$\bar{\chi}^v[\mathbf{n}]$	$\kappa^{sign}[\mathbf{n}]$	$\chi^{flip}[\mathbf{n}]$
1	1	14	1
1	0	13	1
1	-1	12	1
0	1	11	1
0	0	10	1
0	-1	11	-1
-1	1	12	-1
-1	0	13	-1
-1	-1	14	-1

using the intermediate quantities

$$\chi^h[\mathbf{n}] = \chi[n_1, n_2 - 1]\sigma[n_1, n_2 - 1] + \chi[n_1, n_2 + 1]\sigma[n_1, n_2 + 1]$$

$$\chi^v[\mathbf{n}] = \chi[n_1 - 1, n_2]\sigma[n_1 - 1, n_2] + \chi[n_1 + 1, n_2]\sigma[n_1 + 1, n_2].$$

$\chi^h[\mathbf{n}]$ and $\chi^v[\mathbf{n}]$ are then truncated to the range -1 through 1 to form

$$\bar{\chi}^h[\mathbf{n}] = \text{sign}(\chi^h[\mathbf{n}]) \min\{1, |\chi^h[\mathbf{n}]|\}$$

$$\bar{\chi}^v[\mathbf{n}] = \text{sign}(\chi^v[\mathbf{n}]) \min\{1, |\chi^v[\mathbf{n}]|\}.$$

The sign-coding context label, $\kappa^{sign}[\mathbf{n}]$, and the sign-flipping factor, $\chi^{flip}[\mathbf{n}]$, are then given in Table 18.2. The binary symbol, s , that is arithmetic coded is defined as

$$s = \begin{cases} 0 & \chi[\mathbf{n}]\chi^{flip}[\mathbf{n}] = 1 \\ 1 & \chi[\mathbf{n}]\chi^{flip}[\mathbf{n}] = -1. \end{cases}$$

The second pass is the magnitude refinement pass. In this pass, all bits from locations that became significant in a previous bit-plane are coded. As the name implies, this pass refines the

Table 18.3 Context Labels for Magnitude Refinement Coding

$\bar{\sigma}[\mathbf{n}]$	$\kappa^{sig}[\mathbf{n}]$	$\kappa^{mag}[\mathbf{n}]$
0	0	15
0	>0	16
1	—	17

Note: A dash denotes “don’t care.”

magnitudes of the samples that were already significant in previous bit-planes. The magnitude refinement context label, $\kappa^{mag}[\mathbf{n}]$, is selected as described in Table 18.3. In the table, $\bar{\sigma}[\mathbf{n}]$ denotes the value of the significance-state variable $\sigma[\mathbf{n}]$ delayed by one bit-plane. In other words, similar to $\sigma[\mathbf{n}]$, $\bar{\sigma}[\mathbf{n}]$ is initialized to zero at the beginning and set to 1 only after the first magnitude refinement bit of the sample has been coded.

The third and final pass is the clean-up pass, which takes care of any bits not coded in the first two passes. By definition, this pass is a “significance” pass, so significance coding, as described for the significance propagation pass, is used to code the samples in this pass. Unlike the significance propagation pass, however, a run-coding mode may also occur in this pass. Run-coding occurs when all four locations in a column of the scan (see Fig. 18.9) are insignificant and each has only insignificant neighbors. A single bit is then coded to indicate whether the column is identically zero or not. If not, the length of the zero run (0 to 3) is coded, reverting to the “normal” bit-by-bit coding for the location immediately following the 1 that terminated the zero run.

18.3.5 Packets and Layers

Packets are the fundamental unit used for construction of the codestream. A packet contains the compressed bytes from some number of coding passes from each codeblock in one precinct of one tile-component. A packet consists of a packet header followed by a packet body. The packet body contains m_i coding passes from codeblock i in order $i = 0, 1, 2, \dots$, and m_i can be any integer including 0. The number of coding passes can vary from block to block. Any number of passes (including zero) is legal. In fact, zero passes from every codeblock is legal. In this case the packet must still be constructed as an “empty packet.” An empty packet consists of a packet header, but no packet body.

The packet header contains the information necessary to decode the packet. It includes a flag denoting whether the packet is empty or not. If not, it also includes the following: codeblock inclusion information (whether $m_i = 0$ or $m_i > 0$ for each codeblock i); the number of completely zero bit-planes (zero MSBs) for each codeblock; the number of coding passes m_i for each included codeblock; and the number of compressed bytes included for each block. It should be noted that the header information is coded in an efficient and embedded manner itself. The data contained in a packet header supplement data obtained from previous packet headers (for the same precinct) in a way to just enable decoding of the current packet. For example, the number of leading zero MSBs for a particular codeblock is included only in the first packet that contains a contribution from that codeblock.

It is worth reiterating that the bit-plane coding of a codeblock is completely independent of any other codeblock. Similarly, the header coding for a packet in a particular precinct is independent of any other precinct.

A layer is a collection of packets from one tile. Specifically, a layer consists of one packet from each precinct of each resolution of each tile-component of one tile.

The first packet of a particular precinct contains some number of coding passes from each codeblock in the precinct. These coding passes represent some (varying) number of MSBs for each quantized wavelet coefficient index in each codeblock of the precinct. Similarly, the following packets contain compressed data for more and more coding passes of bit-planes of quantized wavelet coefficient indices, thereby reducing the number of missing LSBs (p) in the quantizer indices $q^{(p)}$. We note here that at the end of a given layer l , the number of missing LSBs (p) can be vastly different from codeblock to codeblock. However, within a single codeblock, (p) cannot differ from coefficient to coefficient by more than 1.

From this discussion, it is now understood that a packet provides one quality increment for one spatial region (precinct), at one resolution of one tile-component. Since a layer comprises one packet from each precinct of each resolution of each tile-component, a layer is then understood as one quality increment for an entire tile.

It should also be clear that the quality increment need not be consistent throughout the tile. Since the number of coding passes in a layer (via packets) can vary codeblock by codeblock, there is complete flexibility in “where and by how much” the quality is improved by a given layer. For example, a layer might improve one spatial region of a reduced resolution version of only the grayscale (luminance) component.

18.3.6 JPEG2000 Codestream

The compressed packets in JPEG2000 are organized to form a codestream. In addition to compressed packet data, the JPEG2000 codestream includes information on fundamental quantities such as image size and tile size as well as coding parameters such as quantization step sizes and codeblock size. All the information necessary for decompression of the codestream is signaled to the decoder inside the codestream. The JPEG2000 codestream syntax provides a compact and efficient organization, while enabling the rich feature set of JPEG2000. A simple JPEG2000 codestream is illustrated in Fig. 18.11. The codestream in the figure consists of a main header followed by a sequence of tile-streams. The codestream is terminated by a 2-byte marker called EOC (end of codestream).

The main header provides global information that is necessary for decompression of the codestream. This information includes image and tile sizes, as well as default values for the quantization and coding parameters. In JPEG2000, each header is composed of a sequence of markers and marker segments. A marker is a 2-byte value. A marker segment is composed of a marker followed by a list of parameters. The first byte of every marker is the hexadecimal value **0xFF**.

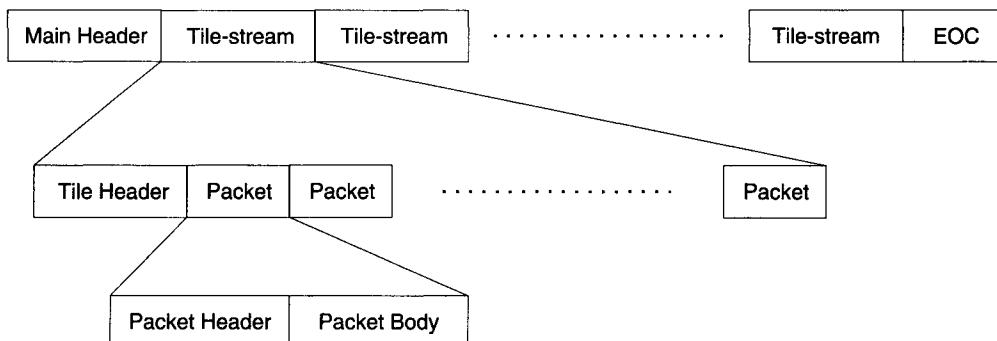
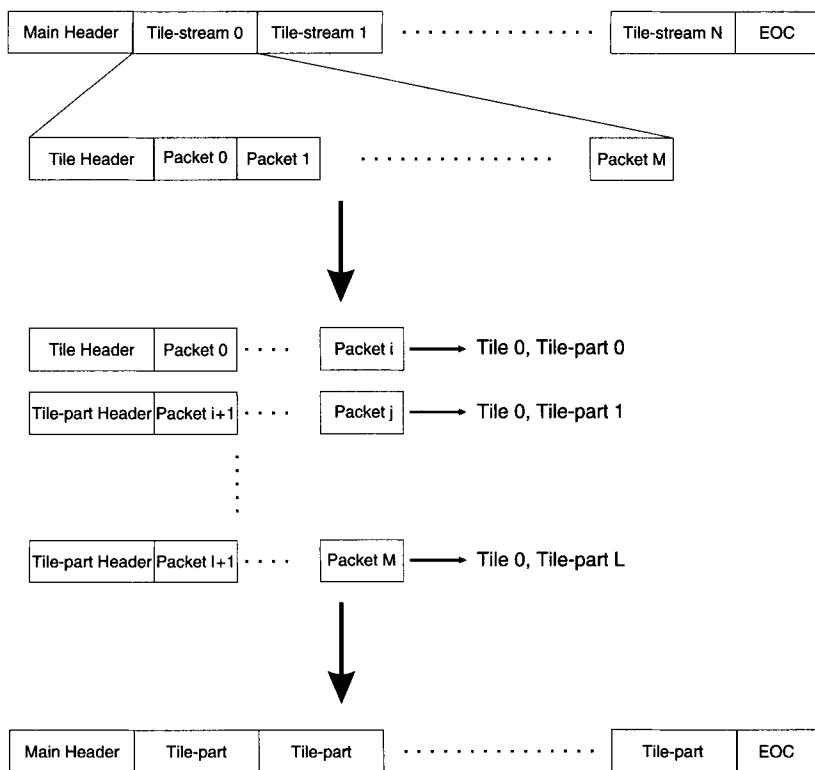


FIGURE 18.11
Simple JPEG2000 codestream.

**FIGURE 18.12**

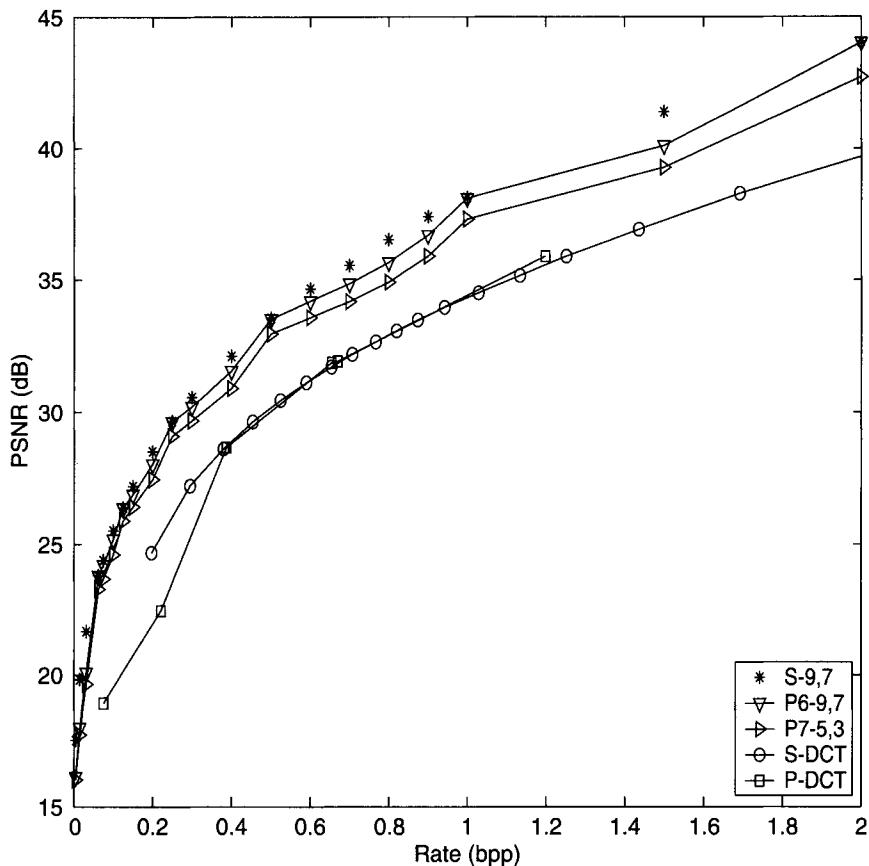
The use of tile-parts in a JPEG2000 codestream.

The tile header contains information that is related to a particular tile. It is possible to use the tile header to overwrite some of the default coding parameters specified in the main header. Furthermore, it is possible to break tile-streams into multiple *tile-parts*. In this case, the first tile-part contains a tile header and the remaining tile-parts contain tile-part headers. Tile-parts allow the progression concepts to be extended to the entire image. A tile-part header is constructed in exactly the same fashion as a tile header. However, only certain marker segments that are allowed in a tile header are allowed in a tile-part header. A JPEG2000 codestream utilizing tile-parts is illustrated in Fig. 18.12.

18.4 PERFORMANCE

Figure 18.13 provides rate-distortion performance for two different JPEG modes and three different JPEG2000 modes for the Bike image (grayscale, 2048 by 2560) from the SCID test set. The JPEG modes are progressive (P-DCT) and sequential (S-DCT), both with optimized Huffman tables. The JPEG2000 modes are single layer with the (9, 7) wavelet (S-9, 7), six-layer progressive with the (9, 7) wavelet (P6-9, 7), and seven-layer progressive with the (5, 3) wavelet (P7-5, 3). The JPEG2000 progressive modes have been optimized for 0.0625, 0.125, 0.25, 0.5, 1.0, and 2.0 bpp and lossless for the 5 × 3 wavelet. The JPEG progressive mode uses a combination of spectral refinement and successive approximation.

The JPEG2000 results are significantly better than the JPEG results for all modes and all bit-rates on this image. Typically JPEG2000 provides only a few decibels of improvement from

**FIGURE 18.13**

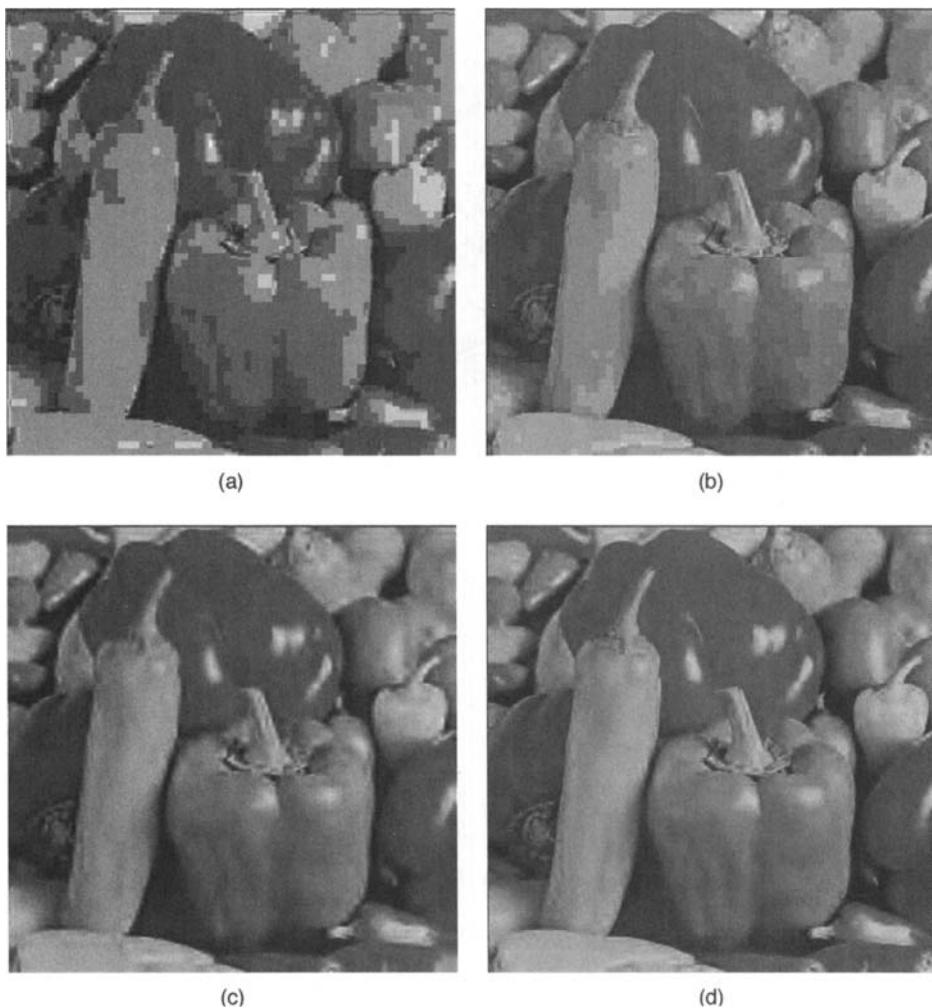
Rate-distortion performances of JPEG and JPEG2000 on the bike image.

0.5 to 1.0 bpp but provides substantial improvement below 0.25 bpp and above 1.5 bpp. It should be noted that the progression in JPEG was not optimized for this image, while the JPEG2000 progressive modes are optimized for the image. However, the ability to perform such optimization in a simple manner is a key advantage of JPEG2000 over progressive JPEG.

With JPEG2000 the progressive performance is almost identical to the single-layer performance at the rates for which the progression was optimized. Once again, this is because the coded data bits do not change. The slight difference is due solely to the increased signaling cost for the additional layers (which changes the packet headers). It is possible to provide “generic rate scalability” by using upward of 50 layers. In this case the “scallops” in the progressive curve disappear, but the overhead increases, so the curve is always lower than the single-layer points.

Figure 18.14 (see also color insert) illustrates the substantial improvement offered by JPEG2000 over JPEG at rates below 0.25 bpp using the Pepper image. In Fig. 14.19, both JPEG and JPEG2000 compressed images at 0.088 and 0.155 bpp are presented. The JPEG images were generated using the sequential mode with optimized Huffman tables, and a single layer was utilized for JPEG2000.

Although JPEG2000 provides significantly lower distortion for the same bit-rate, the computational complexity is significantly higher. Current JPEG2000 software implementations run slower than optimized JPEG codecs by roughly a factor of 3. The speed of the JPEG2000 code should increase over time with implementation optimization.

**FIGURE 18.14**

Peppers image compressed at very low bit-rates using JPEG and JPEG2000. (a) JPEG compressed at 0.088 bpp. (b) JPEG compressed at 0.155 bpp. (c) JPEG 2000 compressed at 0.088 bpp. (d) JPEG 2000 compressed at 0.155 bpp. (See also color insert.)

JPEG2000 also requires more memory than sequential JPEG, but not as much as might be expected. For conceptually simple implementations, encoders and decoders buffer entire codeblocks, typically 64 by 64 for entropy coding. However, block-based or sliding window implementations of the wavelet transform allow operation on just a few codeblocks at a time.

Table 18.4 shows the lossless performance of JPEG, JPEG-LS, and JPEG2000. JPEG uses a predictor and Huffman coding (no DCT). In each case the best of all predictors has been used, and Huffman tables have been optimized. For primarily continuous-tone imagery as in the Aerial2, Bike, and Barbara images, JPEG2000 is close to JPEG-LS and substantially better than JPEG lossless. For images with text and graphics (2/3 of the Cmpnd1 image contains only rendered text), JPEG-LS provides a gain of almost a factor of 2 over JPEG lossless and JPEG2000. Of course, the entire feature set is available for even losslessly compressed JPEG2000 imagery, while the other two algorithms can provide only lossless raster-based decompression.

Table 18.4 Lossless Performance of JPEG, JPEG-LS, and JPEG2000

Method	Images			
	Aerial2	Bike	Barbara	Cmpnd1
JPEG	5.589	4.980	5.663	2.478
JPEG-LS	5.286	4.356	4.863	1.242
JPEG2000 (50 layers)	5.467	4.562	4.823	2.166

18.5 REFERENCES

1. ISO/IEC 15444-1, 2000. JPEG2000 Image Coding System.
2. Taubman, D. S., and M. W. Marcellin, 2002. *JPEG2000: Image Compression Fundamentals, Practice and Standards*, Kluwer Academic, Norwell, MA.
3. ISO/IEC 14495-1, 1999. Information Technology—Lossless and Near-Lossless Compression of Continuous-Tone Still Images.
4. Akansu, A. N., and R. A. Haddad, 1992. *Multiresolution Signal Decomposition: Transforms, Subbands, Wavelets*, Academic Press, San Diego, CA.
5. Vaidyanathan, P. P., 1993. *Multirate Systems and Filter Banks*, Prentice-Hall, Englewood Cliffs, NJ.
6. Vetterli, M., and J. Kovacevic, 1995. *Wavelets and Subband Coding*, Prentice-Hall, Englewood Cliffs, NJ.
7. Strang, G., and T. Nguyen, 1996. *Wavelets and Filter Banks*, Wellesley-Cambridge Press, Wellesley, MA.
8. Cohen, A., I. Daubechies, and J.-C. Feaveau, 1992. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, Vol. 45, No. 5, pp. 485–560, June 1992.
9. Calderbank, R., I. Daubechies, W. Sweldens, and B.-L. Yeo, 1998. Wavelet transforms that map integers to integers. *Applied Computational Harmonics Analysis*, Vol. 5, No. 3, pp. 332–369.
10. Marcellin, M. W., M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner, 2001. An overview of quantization in JPEG2000. *Signal Processing: Image Communication: Special Issue on JPEG2000*, Vol. 17, No. 1, pp. 73–84, December 2001.

This Page Intentionally Left Blank

PNG Lossless Image Compression

GREG ROELOFS

OVERVIEW

PNG, the Portable Network Graphics format, is a robust, extensible, general-purpose, and patent-free image format. In this chapter we briefly describe the events that led to its creation in early 1995 and how the resulting design decisions affect its compression. Then we examine PNG’s compression engine, the deflate algorithm, quickly touch on the zlib stream format, and discuss some of the tunable settings of its most popular implementation. Since PNG’s compression filters are not only critical to its efficiency but also one of its more unusual features, we spend some time describing how they work, including several examples. Then, because PNG is a practical kind of a format, we give some practical rules of thumb on how to optimize its compression, followed by some “real-world” comparisons with GIF, TIFF, and gzip. Finally, we cover some of the compression-related aspects of Multiple-Image Network Graphics, PNG’s animated cousin, and wrap up with pointers to other sources for further reading.

19.1 HISTORICAL BACKGROUND

Image formats are all about design decisions, and such decisions often can be understood only in a historical context. This is particularly true of the Portable Networks Graphics format (PNG), which was created during—and as part of—the “Wild West” days of the Web.

Though PNG’s genesis was at the beginning of 1995, its roots go back much further, to the seminal compression papers by Abraham Lempel and Jacob Ziv in 1977 and 1978 [1, 2]. The algorithms described in those papers, commonly referred to as “LZ77” and “LZ78” for obvious reasons, spawned a whole host of refinements. Our immediate concern, however, is with just one: Terry Welch’s, published in 1984 [3], now known as Lempel–Ziv–Welch or simply LZW.

LZW initially was used mainly in modems and networking equipment, where its speed and efficiency were matched by the ease of its implementation in silicon. It was also implemented in software in the form of the `compress` command on Unix systems, but the most interesting development from our perspective was its incorporation into CompuServe's GIF image format in 1987. Originally used only within CompuServe, GIF's design soon led it to the wide open spaces of Usenet, where its open and portable design gave it an advantage over the proprietary and platform-specific formats in use at the time. By 1991, GIF—true to its name—had indeed become a format for graphics interchange and was the dominant one on the early Internet. Little wonder that it also became one of the first to be tapped for the budding World Wide Web.

Alas, one important detail got overlooked in all the excitement: The LZW algorithm on which GIF was based had actually been patented by Sperry (which later merged with Burroughs to become Unisys). In fact, the LZW patent was well known, but prevailing wisdom at the time was that algorithms, *per se*, could not be patented—only their instantiations in hardware could. Thus, while modem manufacturers were required to pay licensing fees, most believed that software implementations such as `compress` and GIF applications were exempt.

That belief came to an abrupt and painful end in December 1994. After more than a year of private discussions with Unisys, CompuServe announced on December 28, 1994 that developers of GIF-supporting software would henceforth be required to pay licensing fees. As one might imagine, the Internet community grew quite excited over the issue, and one result was the creation of PNG during the first 2 months of 1995.

19.2 DESIGN DECISIONS

We will touch on some of the subsequent events surrounding PNG later in this chapter, but for now we have enough background to understand the things that shaped PNG's design—and how, in turn, that design affects its compression.

First and foremost, PNG was designed to be a complete and superior replacement for GIF. At the most basic level, that meant PNG should satisfy the following requirements:

- fully lossless compression method;
- fully patent-free compression method (!);
- support for lossless conversion from GIF to PNG and back again;
- support for color-mapped images (also known as “indexed-color” or “palette-based”); and
- support for transparency.

But PNG's designers were not satisfied with a minimal replacement. Since essentially *any* change would break compatibility with existing software, the group felt that there was little harm in (and considerable benefit to) doing a more extensive redesign. In particular, there was a strong desire for a format that could be used to archive truecolor images in addition to color-mapped images, yet remain considerably simpler and more portable than the heavyweight TIFF format. “Real” transparency support was also a priority. To this end, the following additional features were included in the list:

- better compression than GIF;
- better interlacing than GIF;
- support for grayscale and truecolor (RGB) modes;
- support for partial transparency (i.e., an alpha channel);

- support for extensions without versioning; and
- support for samples up to 16 bits (e.g., 48-bit RGB).

Finally, a few group members' experience with compression and archiving utilities—particularly with the sorts of problems commonly encountered in transferring such utilities' archives between systems—led to two further requirements:

- support for testing file integrity without viewing; and
- simple detection of corruption caused by text-mode transfers.

We have glossed over one major feature of any proposed GIF replacement: multi-image capability, especially animation. There are two reasons for this. The first is that, historically, multi-image capability was *not* a major feature of GIF; in fact, it was hardly ever used. Indeed, not until September 1995, 6 months after PNG was finalized, did Netscape introduce animated GIFs to the world. The second reason is that the PNG developers decided quite early on to restrict PNG to single-image capability—both for simplicity and for compatibility with Internet standards, which distinguish between still images and animations or video—and later to create a second, related format for animations and other multi-image purposes. Thus was born MNG, the Multiple-Image Network Graphics format, which we discuss at the end of this chapter.

The result of the PNG requirements, hammered out over a period of 2 months, was a clean, simple, component-based format [4]. The fundamental building block of PNG images is the *chunk*, which consists of a type-identifier (loosely speaking, the chunk "name"), a length, some data (usually), and a *cyclic redundancy code* value (CRC). Chunks are divided into two main categories, critical and ancillary, and there are roughly two dozen types defined. But the simplest possible PNG consists of only three critical chunks: an image-header chunk, IHDR, which contains basic info such as the image type and dimensions; an image-data chunk, IDAT, which contains the compressed pixel data; and an end-of-image chunk, IEND, which is the only officially defined chunk that carries no payload. Color-mapped images require one more chunk: PLTE, which contains a palette of up to 256 RGB triplets. (Unlike GIF, the number of palette entries is not restricted to a power of 2.)

In addition to the minimal subset of chunks, all PNG images have one other component at the very beginning: an 8-byte signature that both identifies the file (or stream) as a PNG image and also cleverly encodes the two most common text-file end-of-line characters. Because text-mode transfers of binary files are characterized by the irreversible conversion of such characters, which is guaranteed to destroy virtually any compressed data, the first 8 or 9 bytes of a PNG image can be used to check for the results of such a transfer and even to identify the most likely kind—e.g., from a Unix system to a DOS/Windows system.

Of course, our concern here is compression, not the gory details of interplatform file transfers. But we already know enough about PNG's design to note one effect on compression efficiency. Insofar as a chunk's type, size, and CRC require 4 bytes of storage each, every PNG image contains a minimum of 57 bytes of overhead: the 8-byte signature, 12 bytes per chunk in at least three chunks, and 13 bytes of header information within the IHDR chunk. And as we will see in the sections below on PNG's compression engine, there are at least 9 bytes of additional overhead within the IDAT chunk. Yet even given a minimum of 66 bytes of overhead, PNG's superior compression method more than makes up the difference in most cases, with the obvious exception of tiny images—most commonly the 1 × 1 spacers and "web bugs" used on many web pages.

Another key principle of PNG's design was the *orthogonality* of its feature set. For example, the core compression algorithm is treated as logically distinct from the preconditioning filters (which we discuss in detail below), and both are independent of the image type and

pixel depth.¹ This has clear advantages in terms of creating a clean architecture and simpler, less error-prone implementations, but it does come at a cost. For example, one could imagine that a particular compression scheme might be particularly well suited to palette images but not to grayscale or RGB. For black-and-white images (i.e., 1-bit grayscale), there is no question that both Group 4 facsimile compression and the JBIG algorithm are much better than PNG, but they can be used on deeper images only through the awkward expedient of applying them to individual bit-planes—i.e., 24 times per image in the case of truecolor.² On the other hand, the simple fact of supporting multiple pixel depths and image types provides some benefits, as well. For example, whereas an 8-bit grayscale GIF must store 768 bytes of extraneous palette data, PNG can avoid this and store the image as native grayscale. For web images on a busy site, a savings of three-quarters of a kilobyte, taken many times per second, may have a discernible effect on bandwidth usage.

But these are all relatively minor issues, at least as far as compression is concerned. Aside from losslessness, the design choice with the largest impact on compression was, unquestionably, the one that triggered the entire effort: the requirement to be free of patents. Together with the implicit requirement that the format be usable—i.e., that it could be encoded and decoded “fast enough” and that doing so would require “little enough” memory—this dramatically limited the possible compression algorithms. Obviously LZW was ruled out immediately, as were many of the other Lempel–Ziv derivatives. Arithmetic compression was known to be an efficient approach, but the fastest variants were all patented by IBM and others, and even those were quite slow compared to the LZ methods. Both Huffman and run-length encoding were fast, memory-efficient, and patent-free, but their compression efficiency was unacceptably low. And while there were any number of other compression algorithms being researched in universities around the world, most of them were lossy, and none of them were free of the possibility of “submarine patents,” i.e., the kind that are applied for in secret and only become public knowledge years later, when they are finally granted.

In the end, there were only two serious candidates for PNG’s compression engine: a particular LZ77 variant that was already in wide use and had survived at least one patent review, and a relative newcomer known as Burrows–Wheeler block transform coding (BWT for short, covered in Chapter 7). While there were a few concerns about BWT’s possible coverage by future patents, it did appear to be clean (which still seems to be the case in 2002). But the main problem was that the early implementations available at the time were quite memory-intensive and horrendously slow—roughly four to eight times the memory and two orders of magnitude slower than the LZ77 alternative. Insofar as BWT’s compression was no more than 30% better than the alternative (and often merely equivalent), and given its relative immaturity at the time, it was ultimately rejected.

19.3 COMPRESSION ENGINE

Thus the core of PNG’s compression scheme is a descendant of LZ77 known as *deflate* [5]. The specifics of the format were defined by PKWARE for their PKZIP 1.93a archiver in October 1991, and Jean-loup Gailly and Mark Adler wrote an open-source implementation that first appeared in Info-ZIP’s Zip and UnZip, and subsequently in GNU gzip and the zlib library

¹ There are a few exceptions, however. For example, even though an alpha (transparency) channel is logically separate from the image type, the PNG specification disallows alpha channels in color-mapped images; to compensate, the palette itself may be extended to a table of RGBA values via the optional tRNS chunk. Also, even though an RGB palette is supported in truecolor (RGB and RGBA) images as a means of providing a suggested palette for color-reduction on limited displays, the extended RGBA palette is *not* allowed as a corresponding feature in RGBA images. And, of course, not every bit depth is supported with every image type; there is no such thing as a 6-bit (2-bits-per-sample) RGB PNG, for example.

² Presumably the compression ratio is none too good in this case, either; the correlations (i.e., redundancies) between different bit-planes would be ignored.

(<http://www.zlib.org/>). Deflate is comparable to or faster than LZW in both encoding and decoding speed, generally compresses between 5 and 25% better for “typical computer files” (though 100–400% better is not uncommon in truecolor images), and never expands incompressible data by more than a fraction of a percent,³ but it has a larger memory footprint. Deflate’s compression efficiency tends to be worse than both Burrows–Wheeler and arithmetic schemes by approximately 30%, but the most common implementations of both approaches also require an order of magnitude more time and memory.

In the simplest terms, deflate uses a sliding window of up to 32 KB (32,768 bytes), with a Huffman encoder on the back end.⁴ Recall from Chapter 6 that a “sliding window” is a very literal description of the key idea of LZ77. Encoding involves finding the longest matching string (or at least a long string) in the 32-KB window immediately prior to the “current position,” storing that as a pointer (distance backward) and a length (in bytes), and advancing the current position—and therefore the window—accordingly.

Of course, one should never trust a one-line description of anything, particularly a compression algorithm. The devil, as they say, is in the details. We will not attempt to cover all of the gruesomeness of the deflate spec—the specification itself is the best reference for that—but there are a number of features that are worth pointing out. We will make a note of those that separate deflate from more conventional LZ77 and LZSS implementations.

Deflate limits match-lengths to between 3 and 258 bytes; this has two important consequences. The first is that the maximum conceivable compression ratio, using 1 bit to encode the distance pointer and 1 bit for the (258-byte) length, is 1032:1. There are some assumptions built into this limit, of course; among them is that the ratio of actual compressed data to overhead, such as header and trailer bits, is infinite. In practice, however, better than 1030:1 is achievable, even with the overhead.

The other consequence of the length limits is that there must be some alternate mechanism to encode sequences of less than 3 bytes—particularly single bytes. In order to prime the sliding window and to accommodate bytes in the input stream that do not appear anywhere in the sliding window, the algorithm must be able to encode plain characters, or “literals.” There is nothing particularly difficult or interesting about that, but it does mean that there are three kinds of symbols rather than two: lengths, distances, and literals. These three alphabets are the grist for the Huffman stage of the deflate engine.

Deflate actually merges the length and literal codes into a single alphabet of 286 symbols. Values 0–255 are the literals, 256 is a special end-of-block code, and 257–285 encode the lengths. Astute readers will note that there is something fishy about an algorithm in which 29 values are used to encode 256 possible lengths. Deflate works around this little problem by using some of the 29 values to encode *ranges* of lengths and then appending between 1 and 5 extra bits to specify the precise value. A similar approach is used for the distance alphabet; 30 values encode 32,768 possible distances, with all but four of the codes requiring between 1 and 13 extra bits.

The two alphabets, lengths/literals and distances, are fed to the Huffman encoder and compressed with either fixed (predefined) or dynamic Huffman codes. A subtlety of the latter case is that the codes are allowed to be no more than 15 bits long; in the understated wording of the deflate specification, “this constraint complicates the algorithm for computing code lengths from symbol frequencies.” (For that matter, finding the longest match in the sliding window is more than a little complex. Fortunately, a decoder need not worry about either issue.)

³ Actually, a 1-byte file could expand by 11 bytes. But the 11 bytes are a fixed overhead, so the claim holds for anything larger than a kilobyte or so. LZW, on the other hand, can expand a 200-KB file to 600 KB in some cases.

⁴ PKWARE recently introduced a 64-KB variant, but its compression is typically only a fraction of a percent better than standard (32 KB) deflate.

Since dynamic coding can tailor itself to the input data, it tends to be more efficient for larger data sets. However, unlike the case with fixed codes, dynamic Huffman trees must be explicitly included in the output stream (literal/length tree first, then distance tree). Thus, for smaller input streams, where the overhead of the trees outweighs their improved compression efficiency, fixed codes are better. Because deflate organizes its output into blocks—something at which we hinted earlier, with the end-of-block code—it can alternate between fixed and dynamic Huffman codes as necessary in order to optimize compression.

What about incompressible data, such as data that have already been compressed with deflate or another algorithm? Deflate includes a third option for such cases: store the data as is. Stored blocks can be up to 64 KB in length, although implementations may further limit their size.⁵

19.4 zlib FORMAT

PNG currently specifies a single compression method, deflate.⁶ It further specifies that the deflated data must be formatted as a *zlib* stream, which has two implications. One is that the deflate stream immediately acquires 6 additional bytes: 2 header bytes and 4 trailer bytes. The header bytes encode basic information about the compression method; in the current zlib specification, only deflate is defined, and only with window sizes equal to a power of 2 between 256 and 32,768 bytes, inclusive. The 4 bytes at the end of the stream are the *Adler-32 checksum* of the (uncompressed) input data. The Adler-32 checksum, defined in Section 2.2 of the zlib specification [6], is a simple extension of the 16-bit Fletcher checksum [7] used in telecommunications, and it is faster to compute than a standard 32-bit CRC is.

The other implication of zlib formatting is that it supports the concept of *preset dictionaries*. Instead of starting with an empty sliding window, as is the usual case, the encoder can prime the sliding window with a predetermined dictionary of strings likely to be found in the data. This is signaled by a flag bit in the zlib header and an additional 4-byte identifier before the compressed stream. (The identifier is actually the Adler-32 checksum of the preset dictionary.) Since only the identifier is transmitted, and not the entire dictionary, the benefits for smaller files are potentially great—for example, a 512-byte datastream could be encoded in just a few bits if the dictionary contained a good match for the stream. On the other hand, both encoder and decoder must know ahead of time what the dictionary is; this tends to be a problem for an image format like PNG, in which there are many decoders in existence, none of which have *a priori* knowledge of any such dictionary. However, as we will see in the MNG section later, preset dictionaries *might* play a useful role in a multi-image format, particularly when they are used to store a collection of small icons or an animated cartoon with a limited number of scene changes.

19.5 zlib LIBRARY

Arguably the most common implementation of the zlib format—definitely the most popular for PNG applications—is Jean-loup Gailly and Mark Adler’s zlib library, which originally defined the format. zlib was a reimplementation of Info-ZIP’s deflate codec and was created in early 1995 specifically for use in PNG implementations. (Since then it has found a home in numerous

⁵ Note that the abstract to the deflate specification, in discussing worst-case expansion, implies a limit of 32 KB. This is incorrect for the format, but it is true of the reference implementation.

⁶ This is something that, for compatibility and standardization reasons, is unlikely to change for a number of years or until a significantly better—but still unencumbered—compression algorithm is discovered, whichever is longer.

Table 19.1 zlib Library Compression Levels and Parameters

Level	Lazy Matches	Good Match Length	Nice Match Length	Max Insert Length	Max Chain Length
1	No	4	8	4	4
2	No	4	16	5	8
3	No	4	32	6	32
Max Lazy Match					
4	Yes	4	16	4	16
5	Yes	8	32	16	32
6	Yes	8	128	16	128
7	Yes	8	128	32	256
8	Yes	32	258	128	1024
9	Yes	32	258	258	4096

other arenas, of course, perhaps most notably as part of Java’s `java.util.zip` classes and the related JAR archive format.) Let us take a quick look at a few of the implementation-specific details in zlib.

As was discussed at length earlier, PNG was designed in response to the patent issues surrounding LZW in GIF. What about patent issues with deflate? It should come as no surprise to anyone that it is *possible* to write a spec-compliant deflate encoder that infringes on various patents. In fact, PKWARE, which created the deflate format, has a relevant patent (involving sorted hash tables), and there are several others that also could apply to a deflate encoder. But the key reason deflate was chosen for PNG was that it *can* be implemented *without* infringing on any patents—and without any significant impact on speed and compression efficiency. Indeed, Section 4 of the zlib specification outlines such an approach and says, “Since many variations of LZ77 are patented, it is strongly recommended that the implementor of a compressor follow the general algorithm presented here, which is known not to be patented per se.”

On a much more practical level, the zlib library supports 10 compression settings. The lowest, level 0, is straightforward: it indicates no compression, i.e., that the data are written using deflate’s stored blocks. (Note that the library limits the size of these blocks to 32 KB rather than the 64 KB that the deflate spec allows.) But the other nine levels encode a number of parameters that collectively trade off between encoding speed and compression efficiency. These are summarized in Table 19.1.

Cenk Sahinalp and Nasir Rajpoot described in Chapter 6, Section 6.5.1, the deflate algorithm’s use of lazy evaluation of LZ77 string-matches. Such an approach can compress better than a simple “greedy” algorithm, but at a cost in complexity and encoding time. Only the six highest zlib levels use lazy matching.

The “good match length” and “nice match length” parameters are thresholds for the lengths of already-matched strings beyond which zlib respectively either reduces its search effort for longer ones or quits altogether. (Note that the longest possible match length, 258 bytes, is supported only for zlib levels 8 and 9.) For levels 4–9, the “max lazy match” parameter is similar to the nice match length, except that it applies to the search for a lazy (secondary) match rather than the primary match.

Since levels 1–3 do not use lazy matching, zlib uses another parameter, “max insert length,” in place of the maximum lazy match value; this determines whether a matched string will be added to zlib’s internal dictionary (hash table). Only very short strings will be added, which improves speed but degrades compression.

Finally, the “max chain length” parameter sets the limit for how much of its internal dictionary zlib will search for a (better) match. For compression level 9, zlib will check hash chains with as many as 4096 entries before moving to the next window position; as one might imagine, this has a significant impact on encoding speed.

In addition to the various implicit parameters hidden within zlib's nine compression levels, the library also supports a trio of explicit parameters that affect compression. The simplest determines the size of the sliding window, which is usually 32 KB but (for encoders only) can be any smaller power of 2 down to 512 bytes.⁷ Smaller windows adversely affect compression efficiency, of course, except in the case that the entire uncompressed stream is smaller than the window size, give or take a couple hundred bytes. And compliant decoders are required to support all window sizes.

The second explicit parameter, `memLevel`, determines the size of zlib's hash table and its buffer for literal strings. It defaults to 8, but the maximum value is 9. Larger values are faster (assuming sufficient memory is available) and improve compression efficiency slightly.

The third and final parameter is slightly more interesting, if only because it involves the one part of zlib that was specifically designed for PNG. The *strategy* parameter tunes zlib according to the expected class of input data. The default value (`Z_DEFAULT_STRATEGY`) emphasizes string-matching, which tends to produce the best results for text files and program objects. The `Z_HUFFMAN_ONLY` value disables string-matching entirely and depends only on Huffman coding, which also makes it relatively fast. A third value, `Z_FILTERED`, is designed for “data produced by a filter (or predictor)” — i.e., PNG-like data — and adopts an intermediate approach of less string-matching and more Huffman coding than the default.

19.6 FILTERS

Filters, in the context of PNG, are a means by which pixel data are *preconditioned* before being fed to the core compression engine, resulting in improved compression performance. This fact, which may seem counterintuitive at first, is best illustrated with an example.

Consider the following string of characters, which in this case represents a line of 26 pixel values:

a b c d e f g h i j k l m n o p q r s t u v w x y z

(For simplicity, assume it is a grayscale image, and $a = 1, b = 2, \dots, z = 26$.) Clearly there is a regular pattern here, but to a byte-oriented compressor, the string is completely incompressible—there are no repeated patterns at all. If, however, we make a simple and reversible transformation, replacing each character with the (numerical) difference between it and the character to its left,⁸ the story is quite different:

With this transformation, even a simple run-length encoder can compress the sequence to 2 bytes (i.e., 26, a). As long as the decoder knows both the compression method (RLE) and filtering algorithm (subtraction in the horizontal direction, with appropriate boundary conditions), it can reconstruct the original data exactly.

Of course, this particular example is a special case; most images are less amenable. On the other hand, most images *do* contain regions of relatively slowly varying colors, and the sequence above represents a specific kind of variation that is quite common in computer graphics: it is a horizontal

⁷ In principle the library also supports 256-byte windows, but a long-standing bug renders this setting unusable in zlib releases through version 1.1.3.

⁸ The issue of boundary conditions—i.e., what lies to the left of the leftmost character—is discussed later.

gradient. So it should not require a great stretch of the imagination to see how pixel-filtering can enhance compression.

PNG uses five such filters, collectively referred to as “filter method 0.” (At the time of this writing, there are no other filter methods either defined or planned for PNG, although a refinement added to the MNG specification in late 2000 might find its way into a future major revision of the PNG spec, if and when such a revision occurs.) Each row of an image may have a different filter; an extra byte at the beginning of the transformed row indicates which was selected. We describe the filters in detail below, but first there are some ground rules that apply to all five.

The first rule is that the final result of all filtering arithmetic is output as unsigned 8-bit integers, even though intermediate steps may require larger values and/or signed values. Modulo-256 arithmetic is used to generate the final values; thus –1 and 511 both map to a final value of 255.

The second rule is that PNG filtering algorithms always operate on “like values.” In particular, in an RGB image, the differencing scheme described above would subtract each pixel’s red value from its neighbor’s red value, and similarly green from green and blue from blue. The rationale behind this approach is simply that the alternative of *intrapixel* differencing does not take advantage of “slowly varying colors”; it sees only the largely uncorrelated differences between the component red, green, and blue values. For example, consider a solid row of orange-yellow pixels, each with RGB values of 255, 192, and 0, respectively. While PNG’s like-from-like differencing produces a stream of highly compressible zeros, intrapixel differencing would result in the sequence 255, 193, 64, 255, 193, 64, . . . , which to the deflate engine looks very much like the untransformed row.

The third rule is that filters always apply to *bytes*, not pixels. For low-bit-depth images—for example, 1-bit grayscale, also known as black-and-white—this means that up to eight pixel values are transformed in each atomic filter operation. For images with 16-bit samples, on the other hand, each sample (gray, red, green, blue, or alpha) is subjected to two independent filtering operations, one on the low-order byte and one on the high-order byte. While this approach was chosen for simplicity (and orthogonality) of both encoding and decoding implementations, it does have a price: PNG’s compression of 16-bit-per-sample images is decidedly poorer than that of other schemes that handle 16-bit samples “natively.” And for the low-bit-depth case, it turns out that filtering is rarely beneficial—although, as always, there are exceptions to the rule.

The fourth rule is that in an interlaced image, filtering applies to each interlace pass as if it were a separate image with its own height and width. To understand this issue, one needs to know a little more about PNG’s interlacing scheme and how it differs from GIF. In the GIF case, there are four passes. In the first pass, every eighth row is transmitted; in the second, the interval is again every eighth row, but offset by four; in the third pass, every fourth row is sent, evenly centered between the previous rows; and in the fourth pass, every other row is transmitted, completing the image. (Simple addition should convince you even if the algorithm is not quite obvious: $1/8 + 1/8 + 1/4 + 1/2 = 1$.) The idea is to present the user with an overall impression of the image very quickly and then fill in the gaps as quickly as possible. While there is a small cost in compression efficiency due to the mixing of different rows and reduction in interrow correlation, the overall effect is that the image is *perceived* to download faster or at least to be more quickly recognizable—and perhaps usable.

But the GIF approach has an unnecessary limitation: it interlaces in only one dimension. Horizontally, the image is at full resolution from the very beginning, and the result in most browsers is that early passes appear extremely stretched in the vertical direction (by up to a factor of 8). PNG’s approach is to interlace both horizontally and vertically, in a total of seven passes, with a stretch factor of 2 or less at all times. But since different passes will then have different widths, and since some filter types apply to pairs of rows, such operations are conceptually much simpler if each pass is treated as a completely separate image for the purposes of filtering.

Now let us have a closer look at the filters themselves.

Of the five filters defined in the current PNG spec, the simplest is the *None* filter, which implements the trivial case of no filtering. As we noted above, the None filter is usually the best choice for grayscale and palette images that are less than 8 bits deep, but for palette images it turns out to be the best *at 8 bits*, also.

The next simplest filters, *Sub* and *Up*, are conceptually similar. Both are differencing filters, like the alphabetic example given earlier; Sub operates on the previous “corresponding byte” (i.e., of the pixel or pixels to the left), while Up operates on the corresponding byte in the previous row of the image. The region outside the image is considered to be filled with zero bytes, so there is never a problem using Up on pixels in the top row or Sub on the first pixel of any row. The PNG specification uses the following notation:

$$\begin{aligned} \text{Sub}(x) &= \text{Raw}(x) - \text{Raw}(x-bpp) \\ \text{Up}(x) &= \text{Raw}(x) - \text{Prior}(x) \end{aligned}$$

Here bpp is the number of bytes per pixel, rounded up to one in the case of fractional values. Thus for a standard 24-bit RGB image, bpp = 3, while for a 2-bit (four-color) color-mapped image, bpp = 1. (The maximum value of bpp is 8, for a 64-bit RGBA image.) Raw(x) is the unfiltered byte at position x, while Raw(x-bpp) is the corresponding byte to the left (Filtering Rule 3) and Prior(x) is the corresponding byte in the previous row. Sub(x) and Up(x) are the filtered bytes, of course, and the subtraction is performed modulo 256 (i.e., as unsigned bytes). The decoder can easily reverse the filtering (again using modulo-256 arithmetic), since at each point in the output stream it knows the filtered byte at the current position and the raw (decoded) bytes at all previous positions:

$$\begin{aligned} \text{Raw}(x) &= \text{Sub}(x) + \text{Raw}(x-bpp) \\ \text{Raw}(x) &= \text{Up}(x) + \text{Prior}(x) \end{aligned}$$

The fourth filter type, *Average*, is basically a combination of Sub and Up. That is, instead of subtracting *either* the value to the left *or* the one above, it subtracts the average of the two:

$$\text{Avg}(x) = \text{Raw}(x) - (\text{Raw}(x-bpp) + \text{Prior}(x)) / 2.$$

The only subtlety is that the sum of Raw(x-bpp) and Prior(x) is performed without overflow (that is, *not* modulo 256), and the division is the integer variety, wherein any fractional parts are truncated (or, alternatively, any remainder is dropped). Again, decoding is straightforward:

$$\text{Raw}(x) = \text{Avg}(x) + (\text{Raw}(x-bpp) + \text{Prior}(x)) / 2.$$

The fifth and final PNG filter type is the most complex and least intuitive of the set, although in absolute terms it is not particularly tricky. It is called the *Paeth* filter, after Alan W. Paeth [8], and it involves not only the pixels to the left of and above the current position, but also the pixel to the upper left. The basic idea is to make an “ideal” prediction (also known as the *initial estimate*) of the current byte value based on a simple linear function of the other three; approximate the ideal prediction by selecting the closest of the other three byte values; and then, as with the other filters, subtract this approximate value (the actual Paeth predictor) from the byte value of the current position. The following pseudocode, adapted from that in version 1.2 of the PNG specification, is a more precise description of the predictor algorithm:

```
function PaethPredictor (a, b, c)
begin
    ; all calculations use full-precision, signed arithmetic
    ; a = left, b = above, c = upper left
    est := a + b - c      ; initial estimate
```

```

da := abs(est - a)      ; distances to a, b, c
db := abs(est - b)
dc := abs(est - c)
; return nearest of a,b,c, breaking ties in order a,b,c.
if da ≤ db AND da ≤ dc then return a
else if db ≤ dc then return b
else return c
end

```

The filtered value is then given by:

$$\text{Paeth}(x) = \text{Raw}(x) - \text{PaethPredictor}(\text{Raw}(x-\text{bpp}), \text{Prior}(x), \text{Prior}(x-\text{bpp}))$$

And, as with the other filters, it may be easily inverted by the decoder to produce the original value:

$$\text{Raw}(x) = \text{Paeth}(x) + \text{PaethPredictor}(\text{Raw}(x-\text{bpp}), \text{Prior}(x), \text{Prior}(x-\text{bpp}))$$

But the Paeth predictor is perhaps best understood through examples. Consider the following pixel values (which may be actual pixels in a grayscale image, just the green values in an RGB image, or some subset of the high bytes in a 64-bit RGBA image):

20	20	c	b
20	20	a	d
ideal estimate = 20 + 20 - 20 = 20			
PaethPredictor = 20			
filtered value = 20 - 20 = 0			

This is the most trivial case: all values are identical, so the prediction should be flawless. The lower right value, d (in boldface), is the one we are currently filtering. The initial (or ideal) estimate is 20, and its distance from each of the three pixel values is the same (zero). Thus the approximate estimate is the same as the ideal one, and the final filtered byte is zero.

Now consider the slightly more interesting case in which there is a horizontal linear gradient:

18	20	c	b
18	20	a	d
ideal estimate = 18 + 20 - 18 = 20			
PaethPredictor = 20			
filtered value = 20 - 20 = 0			

Once again, both the ideal estimate and the Paeth predictor give the same value, 20, which results in a filtered value of zero. The vertical case is similar.

The 45° diagonal gradients are the next step up in complexity:

20	22	c	b
18	20	a	d
ideal estimate = 18 + 22 - 20 = 20			
PaethPredictor = 20			
filtered value = 20 - 20 = 0			

```

22 20          c  b
20 18          a  d
ideal estimate = 20 + 20 - 22 = 18
PaethPredictor = 20
filtered value  = 18 - 20 = -2 = 254 (mod 256)

```

In the first case the gradient is along the a–b diagonal, and as before, the prediction is exact. But in the second case, the gradient is along the c–d diagonal, and while the initial estimate is an exact match, the Paeth predictor is not—resulting in a filtered value of either 2 or 254, depending on the direction of the gradient. We will comment further on this below.

Our final gradient examples are still linear, but they are oriented (more) arbitrarily with respect to the natural axes of the bit-mapped image:

```

18 19          c  b
32 33          a  d
ideal estimate = 32 + 19 - 18 = 33
PaethPredictor = 32
filtered value  = 33 - 32 = 1

18 20          c  b
32 34          a  d
ideal estimate = 32 + 20 - 18 = 34
PaethPredictor = 32
filtered value  = 34 - 32 = 2

```

As in the previous examples, the ideal estimate turns out to be exact—it is, after all, a linear estimate—while the Paeth predictor is off by one or two.

Finally, consider a pair of “worst case” examples, at least from the perspective of the initial estimate’s range:

```

0 255          c  b
255 0          a  d
ideal estimate = 255 + 255 - 0 = 510
PaethPredictor = 255
filtered value  = 0 - 255 = -255 = 1 (mod 256)

255 0          c  b
0 255          a  d
ideal estimate = 0 + 0 - 255 = -255
PaethPredictor = 0
filtered value  = 255 - 0 = 255

```

These represent the two possible cases for a checkerboard arrangement of pixels, and neither the “ideal” estimate nor the Paeth predictor is a good prediction. Indeed, the resulting byte stream alternates between 1 and 255, which, from the compression engine’s perspective, is essentially identical to the unfiltered byte stream.

All of these examples prompt an obvious question: given that the ideal estimate seems to be a better predictor than the Paeth approximation, why not simply use the ideal one instead? As the last examples showed, the ideal estimate can vary between –255 and +510, but the filtering operation subtracts using modulo-256 arithmetic anyway, so there is no harm in that.

In fact, there are two reasons. The first is intrinsic to the design of the filter; it was created to predict smoothly varying colors. But if the byte values are near their maximum, for example, it

is possible for the ideal estimate to wrap around to the minimum value instead:

253 254	c b	
255 254	a d	
$\text{ideal estimate} = 255 + 254 - 253 = 256 \equiv 0 \pmod{256}$		
PaethPredictor = 255		

Of course, since the filtering operation is performed with modular arithmetic, there is little difference in the final byte stream (254 vs 255 in this example), so this is a weak reason, at best. The second reason is more substantial: there will be many more possible filtered values with the ideal estimate—which need not match any of the input bytes—than with the Paeth estimate. For example, if one has an RGB image in which only four colors are used, there are, at most, 37 possible values for the filtered bytes: zero (when the predicted value equals the target value), plus 4×3 nonzero possibilities per color channel. Using the ideal estimate would produce up to 163 possible values for the filtered bytes, almost four and a half times as many. Paeth’s claim was that the larger alphabet of byte values would reduce the compressibility of the filtered stream, which is likely. On the other hand, the greater propensity for exact matches would produce an output stream with more zeros, which should improve compression—at least for some images. To our knowledge, this has never actually been tested in the context of PNG, so it might be an interesting experiment to try.

We will close our discussion of the Paeth filter with a quick look at one of the more famous examples of its use, a $512 \times 32,768$ RGB image containing one pixel of every possible 24-bit color. The name of the image is `16million`—logically enough—and its uncompressed size is 48 MB. Standard deflate (`gzip`, in this case) reduces it to just under 36 MB. But the Paeth-filtered PNG (with the exception of the top row, which uses the Sub filter) rings in at 115,989 bytes, an overall compression factor of 434:1 and more than 300 times better than the unfiltered approach.⁹

19.7 PRACTICAL COMPRESSION TIPS

Knowing the mechanics of PNG’s five compression filters and seeing their effects on one (fairly contrived) example is only the beginning, however. In fact, arguably the most interesting aspect of filtering is determining which filter types to use. A simple counting argument shows that an exhaustive “brute force” approach is untenable for all but the smallest images: with a choice of five filters for each row, the total number of possibilities is $5^{\text{image_height}}$, which comes to nearly 10 million for a 10-row image and almost 10^{70} for a 100-row image. To get a handle on this, consider that at a rate of 3000 possibilities per second—which, keep in mind, means that the *entire image* is being test-compressed 3000 times per second—a computer would be occupied for the entire current age of the universe (ten billion years) checking all possible filter combinations for a single, 30-row image. And that does not even cover the different zlib compression levels and strategies, which would extend the duration by a relatively paltry additional factor of 27.

So brute force is Right Out. Instead, the PNG developers experimented on different image types and came up with some rules of thumb that are simple, fast, and (apparently) close to optimal, at least in most cases. The first rule of thumb is simple: use filter type None for all palette images and all sub-8-bit grayscale images. Byte-level filtering simply does not work very well when each byte contains multiple pixels, and in the case of 8-bit color-mapped images, the palette indices (which constitute the actual pixel data) are usually quite uncorrelated spatially, even though the

⁹ A 4096×4096 variant compresses to 59,852 bytes, thanks to the longer runs of uninterrupted pixels. (Recall that the filter type is encoded as a byte at the beginning of each row; its value is 4 for Paeth.) Obviously these images are highly unusual, but the point remains: Filtering can have a dramatic effect on compression efficiency.

colors they represent may vary smoothly. Color-mapped images are also often subject to dithering, which amounts to a compression-destroying partial randomization of the pixels.

The second rule of thumb is also fairly simple: Use adaptive filtering for all other PNG image types. More specifically, for each row choose the filter that *minimizes the sum of absolute differences*. What this means is that, rather than using modulo-256 arithmetic when subtracting the predicted value from the actual value at each pixel position, instead do so using standard (signed) arithmetic. Then take the absolute value of each result, add all of them together for the given row, and compare to the sums for the other filter types. Select the filter produces the smallest sum of absolute differences and proceed to the next row.

On the face of it, the absolute-differences heuristic may not seem particularly good. Among other things, it is biased *against* the None filter for relatively light images (i.e., with color values near 255) but *in favor* of it for dark images. The PNG specification even contains the caveat that “it is likely that much better heuristics will be found as more experience is gained with PNG.” Yet in the 8 years since PNG was created, no better approach has been found, and the test results in the next section suggest that there may not *be* a significantly better heuristic in the case of most larger images. Still, one never knows.

Both of these rules of thumb are implemented in `libpng` [9], the free reference library used by most PNG-supporting applications, so neither application developers nor users *should* need to worry about them in most cases. Sadly, there are a few well-known applications whose programmers overlooked the *Recommendations for Encoders* section of the PNG specification, with the unfortunate result that they expose their users to the complexity and unpredictability of PNG filter selection but fail to provide any way to adjust the zlib compression level—the one parameter that is simultaneously the most intuitive and the most effective means of trading off compression efficiency for encoding speed. Users of such applications will need to keep the first two rules in mind or else rely on a third-party utility to improve the compression of their PNG images.

If the adaptive approach of Rule Two is too computationally expensive (for example, if implemented in an embedded device), then the Paeth filter is recommended instead. As we have seen, it performs well when linear gradients are involved.

The third rule of thumb has to do with packing pixels and may seem slightly counterintuitive, given the first rule’s stricture about not filtering low-bit-depth images. If filtering is ineffective for such images, why not first expand the 1-, 2-, or 4-bit data to 8 bits (i.e., 1 byte per pixel) and then filter and compress the result for a net gain? As it turns out, such an approach is rarely effective—although there are always exceptions to the rule. In general, though, the initial expansion by a factor of 2, 4, or 8 is almost never recovered by the deflate engine, much less exceeded. Packing as many pixels as possible into each byte remains the best approach in most cases.

There are other rules of thumb for PNG compression, but for the most part they amount to no more than common sense. For example, as in many endeavors, *use the best tool for the job*. If your image is photographic and absolute losslessness is not required, then by all means use JPEG or an equivalent lossy method! It will be far more efficient than any lossless approach. On the other hand, if the image contains only a handful of colors, standard JPEG is almost never appropriate; the implicit initial conversion from, say, 2-bit pixels to 24-bit pixels will overwhelm even JPEG’s lossy compression algorithm—and on top of that, the results most likely will look horrible. Also, avoid converting JPEG images to PNG whenever possible. The bit-level effects of JPEG’s lossy algorithm are quite detrimental to PNG’s row filters and deflate engine. Even worse, a low-bit-depth image improperly stored in JPEG format will acquire many more colors than it started with, requiring deeper pixels in PNG format and further degrading compression.

Within the realm of PNG image types, be aware of the differences and, again, use the best one for the job. Never store an image as 24-bit RGB if an 8-bit palette will do, and do not use a palette if grayscale will suffice. Avoid interlacing if it is not useful (such as in small images or those not intended for Web use); PNG’s two-dimensional scheme is even more harmful to compression than GIF’s one-dimensional approach is. And it should go without saying that an alpha channel

Table 19.2 Waterloo GreySet1 (256 × 256)

Image Name	GIF (bytes)	Standard PNG		“Optimal” PNG			Filters	Compression Level	Strategy
		(bytes)	vs GIF	(bytes)	vs GIF	vs PNG			
Bird	47,516	32,474	-31.7%	31,448	-33.8%	-3.2%	All	2	Huffman
Bridge	76,511	48,511	-36.6%	48,451	-36.7%	-0.1%	All	9	Filtered
Camera	55,441	38,248	-31.0%	38,087	-31.3%	-0.4%	All	2	Huffman
Circles	1,576	1,829	+16.1%	1,117	-29.1%	-38.9%	0	9	Filtered
Crosses	1,665	2,000	+20.1%	1,365	-18.0%	-31.8%	2	9	Default
Goldhill	68,450	44,941	-34.3%	44,850	-34.5%	-0.2%	All	2	Huffman
Horiz	683	693	+1.5%	297	-56.5%	-57.1%	0	9	Filtered
Lena	71,115	41,204	-42.1%	41,142	-42.1%	-0.2%	All	4	Filtered
Montage	42,449	24,096	-43.2%	23,848	-43.8%	-1.0%	All	9	Filtered
Slope	29,465	11,683	-60.3%	11,204	-62.0%	-4.1%	2	9	Filtered
Squares	931	640	-31.3%	181	-80.6%	-71.7%	0	9	Default
Text	4,205	2,339	-44.4%	1,126	-73.2%	-51.9%	All	9	Filtered
Total size	400,007	248,658	-37.8%	243,116	-39.2%	-2.2%			

is a complete waste of space in an opaque image—yet there are more than a few images floating around the Internet with precisely that problem.

Finally, be wary of misleading comparisons. A 24-bit image saved in PNG or TIFF formats will be saved losslessly at 24 bits 99% of the time; the same image stored in GIF format will, of necessity, be reduced to an 8-bit file. It should be no surprise to anyone that the GIF will be smaller in this case—after all, it was saved using what amounts to a lossy compression method.

19.8 COMPRESSION TESTS AND COMPARISONS

Speaking of comparisons, we have discussed compression theory and the details of its implementation at moderate length but have yet to show much in the way of numbers and real-life results. Since we began the chapter with a discussion of PNG’s origins in the GIF/LZW controversy, we will first compare the two formats directly. We will use the *Waterloo BragZone Répertoire*¹⁰ for all of the comparisons; it is a set of 32 grayscale and color images with a good combination of natural, synthetic or “artistic,” and mixed subject matter. In the case of GIF, however, we will necessarily limit the comparison to just the 24 grayscale images, since GIF realistically cannot support more than 256 colors (or shades of gray) without loss.¹¹

Table 19.2 shows the results of GIF and PNG compression on the Répertoire’s GreySet1, which is composed entirely of 256 × 256 images. Raw values are the byte sizes of the image files, rather than just the LZW or filtered-deflate sizes—this chapter, after all, is devoted to an image format, with all of its features and overhead, and not merely a compression method. One of the first things to notice is that GIF/LZW actually *expands* three of the images. A 256 × 256, 8-bit, uncompressed grayscale image requires precisely 65,536 bytes (plus perhaps a few hundred bytes of overhead for the file format), but *bridge*, *goldhill*, and *lena* are larger than that by 10975, 2914, and 5579

¹⁰ <http://links.uwaterloo.ca/bragzone.base.html>.

¹¹ In principle, GIF *can* support more than 256 colors, but only by using multiple images (either by segmenting an image into rectangular tiles or by defining an appropriate set of transparent overlays), each of which carries its own palette and is therefore limited to no more than 256 colors. In the worst case, this would require more than 1024 bytes per 256 pixels (768 bytes for the palette, 256 bytes for the pixel indices, and a few bytes of overhead for the image block), resulting in an overall expansion factor of 33% over uncompressed RGB data. In addition, it is unclear whether more than a handful of GIF decoders can display such images. As always, use the right tool for the job.

bytes, respectively. This is a characteristic of standard LZW implementations, which have been observed to expand extreme cases by a factor of 3 or more. An optimal LZW encoder need never expand its input by more than a factor of 9/8 or so (12.5%), however.

The *standard PNG* column is the result of a straight conversion to PNG with standard filter heuristics, in this case using the `gif2png` [10] utility. PNG significantly outperforms GIF/LZW overall, but it is larger on three of the synthetic images (*circles*, *crosses*, *horiz*). This is because the PNGs were created as 8-bit grayscale images, while the corresponding GIFs all have depths of 3 bits or less. The “*optimal*” PNG column is the result of the `pnmtopng` [11] utility—which is smart enough to use a gray palette for low-bit-depth images—and of the `pngcrush` [12] utility, which explores (in this case) many combinations of PNG row-filters, zlib compression levels, and zlib strategies. The word “many” is important and is why “*optimal*” is in quotes; as we noted earlier, a complete search of the parameter space would take eons, at least with anything short of a full-blown quantum computer. In the case of `pngcrush`, six filter combinations are checked: five in which one PNG filter type is used for the entire image, and one adaptive strategy using the “absolute differences” heuristic on each row. Together with the various zlib compression settings, it tests up to 120 different possibilities.¹²

The “*optimal*” results are interesting in several respects. For example, the five images showing the most significant improvement (more than 30%) are the smallest; i.e., they are already the most compressed. They also happen to be the only ones for which sub-8-bit, palette-based encoding is possible, and together with *slope*, they are the only purely synthetic images in the set. Also note that only two of the images, *crosses* and *squares*, use zlib’s default (gzip-like) strategy, while three of the photographic images compressed best with the Z_HUFFMAN_ONLY setting. The latter three are also the only ones for which a very low zlib compression level (2) was used, but this is slightly misleading: while the Z_HUFFMAN_ONLY strategy could, in principle, show different behavior depending on whether or not lazy matching is used, empirically its effects are entirely independent of the compression level. Finally, note that adaptive filtering (using the standard PNG heuristic) is preferred in most of the images, followed by no filtering (filter type 0). The “*up*” filter (type 2) worked best in two of the synthetic images, however.

Table 19.3 shows the corresponding results on the Repertoire’s GreySet2, which consists of eight 512×512 images and four odd sizes (one smaller, three larger). Unlike the previous set, only one image in this set is completely synthetic (*france*), and one image is a Web-like montage of photographs separated by white space and adorned with short captions (*library*). Also unlike the first set, all 12 PNGs are 8-bit grayscale, although two of the GIFs are 7 bits (*frog* and *mountain*) and one is 6 bits (*washsat*). (Recall that the next lowest depth supported by PNG is 4 bits.)

The most interesting feature of this set of results is that the Z_HUFFMAN_ONLY zlib strategy is most effective in seven of the cases, while the Z_FILTERED strategy works best in only one. The other four cases, for which the default zlib strategy proved most effective, are also those for which filtering was least effective (that is, filter type None worked best). It is also worth noting that, although GIF outperformed the standard PNG conversion in three cases, “*optimal*” PNG compression won in every case, with an overall 33% improvement over GIF/LZW.

The results from the first two sets of images clearly demonstrate that different combinations of zlib compression parameters and PNG filters can have a dramatic effect on overall compression efficiency, but what is not clear is how much of the effect is due to each degree of freedom. In Table 19.4 we take a somewhat different approach: Instead of simply comparing against standard LZW results (in this case TIFF/LZW, since the final set consists of 24-bit RGB images), we also include the results of ordinary, unfiltered deflate compression—specifically, using the raw PPM format of Jef Poskanzer’s PBPLUS utilities, with standard `gzip` providing the deflate compression.

¹² This is for the `-brute` (force) option of `pngcrush`. The utility’s default mode checks only five or six possibilities and almost always comes within half a percent of the brute-force result. But here we were interested in achieving the best possible results without regard for the time required.

Table 19.3 Waterloo GreySet2 (Various Sizes, Mostly 512 × 512)

Image Name	GIF (bytes)	Standard PNG		“Optimal” PNG			Filters	Compression Level	Strategy
		(bytes)	vs GIF	(bytes)	vs GIF	vs PNG			
Barb	287,781	173,773	-39.6%	173,374	-39.8%	-0.2%	All	2	Huffman
Boat	236,608	151,914	-35.8%	150,639	-36.3%	-0.8%	All	2	Huffman
France	33,282	17,571	-47.2%	14,503	-56.4%	-17.5%	0	9	Default
Frog	160,739	231,993	+44.3%	148,302	-7.7%	-36.1%	0	9	Default
Goldhill	255,937	160,151	-37.4%	159,458	-37.7%	-0.4%	All	2	Huffman
Lena	264,422	150,926	-42.9%	150,596	-43.0%	-0.2%	All	4	Filtered
Library	113,658	104,873	-7.7%	96,594	-15.0%	-7.9%	0	9	Default
Mandrill	306,690	204,063	-33.5%	203,744	-33.6%	-0.2%	All	2	Huffman
Mountain	233,975	253,550	+8.4%	202,946	-13.3%	-20.0%	0	9	Default
Peppers	265,732	158,764	-40.3%	158,260	-40.4%	-0.3%	All	2	Huffman
Washsat	87,800	105,784	+20.5%	74,701	-14.9%	-29.4%	4	2	Huffman
Zelda	251,370	139,395	-44.5%	138,464	-44.9%	-0.7%	All	2	Huffman
Total size	2,497,994	1,852,757	-25.8%	1,671,581	-33.1%	-9.8%			

Table 19.4 Waterloo ColorSet (Various Sizes, 512 × 512 to 1118 × 1105)

Image Name	TIFF (bytes)	gzip’d PPM		“Optimal” PNG			Filters	Compression Level	Strategy
		(bytes)	vs TIFF	(bytes)	vs TIFF	vs gzip			
Clegg	1,736,820	518,240	-71.2%	484,646	-72.1%	-6.5%	All	9	Filtered
Frymire	758,358	252,448	-66.7%	251,616	-66.8%	-0.3%	0	9	Default
Lena	950,128	733,327	-22.8%	475,487	-50.0%	-35.2%	All	4	Filtered
Monarch	1,215,930	846,772	-30.4%	615,329	-49.4%	-27.3%	All	9	Filtered
Peppers	911,336	678,298	-25.6%	425,617	-53.3%	-37.3%	All	2	Huffman
Sail	1,272,498	952,235	-25.2%	777,176	-38.9%	-18.4%	3	9	Default
Serrano	315,432	107,181	-66.0%	106,419	-66.3%	-0.7%	0	9	Default
Tulips	1,373,006	1,004,650	-26.8%	680,950	-50.4%	-32.2%	All	9	Filtered
Total size	8,533,508	5,093,151	-40.3%	3,817,240	-55.3%	-25.1%			

As before, there are a number of interesting features in these results. For one thing, even ordinary `gzip` (compression level 9) performs much better than TIFF/LZW—three times better in several cases, in fact, and 40% better overall. But when we add PNG filtering on top of that, we lop off another 25% over unfiltered `gzip` or an additional 15 percentage points relative to TIFF. In the cases with the largest gains, adaptive filtering and the Z_FILTERED zlib strategy usually worked best, while the two cases with the smallest gains used (unsurprisingly) exactly the same parameters as `gzip`: no filtering, maximal compression level, and default zlib strategy. In fact, the three images with the smallest differences between PNG and `gzip`—*clegg*, *frymire*, and *serrano*—are also the three “artistic” images; the other five are photographs.

One could devote considerable effort to statistical analysis of the filters used in the adaptive cases, how those choices interact with the deflate engine (or an alternative, such as the BWT-based `bzip2`), and whether, given the existing set of five PNG filter types, one might derive a better heuristic for adaptive filtering. But that is considerably beyond the scope of this chapter. It is worth noting again, however, that in the 8 years since the PNG specification was created, no better general-purpose heuristic than the “minimal sum of absolute differences” has ever been found. Indeed, even the somewhat computationally intensive approach used by `pngcrush`, wherein

either 5 or 120 combinations of compression parameters are tried, usually manages no more than 1% improvement over the standard PNG heuristic.¹³

19.9 MNG

MNG, short for *Multiple-Image Network Graphics*, was originally conceived as a minimal metaformat for gluing together sequences of PNG images. But because there was no pressing need for a multi-image or animated PNG extension in early 1995, its development was left on hold for more than a year, allowing developers time instead to write the first PNG implementations and to begin the long process of standardization.

In the meantime, as we noted earlier, Netscape introduced animated GIFs in late 1995, and by early 1996 it was abundantly clear that they were a very popular feature and likely to remain so for the foreseeable future. As a result, in May 1996 the PNG group began discussing MNG once more, in hopes of hammering out a PNG-like specification within a PNG-like time frame—more or less. However, unlike the PNG case, in which there was broad consensus on the main requirements for the format almost from the very beginning, the MNG feature set was the subject of considerable disagreement. Proposals ranged from the minimalist “glue” format envisioned in the beginning to a full-blown, audio/video format approaching MPEG in complexity. In addition, MNG simply was not as interesting as PNG to many of the original developers; for most applications, including print, a still image is far more useful than a moving one, and a moving image that makes noise can be downright annoying.

As a result, MNG development was exceedingly slow, even by international standards. In the end, a middle ground between the minimalist approach and the high end was found, but in truth, this was due more to attrition than to a true consensus of all involved. Nevertheless, the final MNG specification [13], approved in January 2001, is a good, solid design that should keep both developers and designers busy for some years to come.

In brief, MNG is a proper superset of PNG designed for animation and other multi-image purposes. As such, its chunk structure and compression engine are essentially identical to those of PNG (with one major exception, on which we touch at the end), so we will not cover them in detail. But there at least half a dozen important differences that can improve MNG’s compression relative to a collection of individual PNG images, and these are certainly worthy of some discussion.

The first is simply the sharing of image information at the chunk level, typically a global palette in color-mapped images. Particularly for collections of small images such as icons, if all or most of the images use the same palette, then eliminating all but one copy of it can be quite beneficial. Just how beneficial depends entirely on the nature of the images, of course, but keep in mind that palettes are uncompressed and can be as large as 780 bytes.

The second improvement over independent PNGs also involves sharing data between subimages, but at a much more intimate level. MNG supports the concept of image *deltas* or differences, essentially subtracting one image from the next and encoding the result.¹⁴ Insofar as this is one of the most complex features of the MNG specification (as defined by the MNG complexity profile), at the time of this writing there is still very little software available to test MNG delta-coding. However, the MNG version of 16million.png uses this technique and reduces the file size

¹³ A completely exhaustive search of the filter/zlib parameter space for a set of 7 short images (5–10 rows each, widths from 10 to 555 pixels) suggests that pngcrush’s heuristics are very nearly optimal. pngcrush found an optimal parameter set in 5 of the cases (more than one set of parameters may result in the same file size), and in the remaining two cases it was larger by only two bytes (0.71% and 2.20%, respectively).

¹⁴ In this sense, it is an *interimage* filtering method similar to the Sub and Up filter types; imagine the images stacked on top of each other, and the three filters correspond to the three principal axes or dimensions. One could also view it as a lossless analogue to standard MPEG-1 techniques.

from 115,989 to 472 bytes, an additional 246:1 improvement over PNG and a fairly impressive 106,635:1 total compression ratio (relative to the uncompressed image size of 50,331,648 bytes).

The third difference is yet another twist on the data-sharing theme: object reuse. Unlike GIF, MNG supports the concept of image objects or “sprites.” Rather than repeatedly encoding the same image data in order to move a section of the image, MNG simply encodes it once, assigns it an object number (index), and thereafter references only the index for moves and copies. In other words, this is yet another example of dictionary-based compression, but on a more macroscopic scale. The potential file-size savings over a corresponding GIF animation are enormous, but they are likely to be realized only if the animation is created as a MNG. Automatic conversion of a repeated-subimage GIF into a object-based MNG is certainly possible, but it would require more intelligence than most conversion programs possess.

Loops are another important difference from PNG, and the ability to nest them gives MNG a substantial advantage over GIF—though at a cost in complexity, too, of course. `16million.mng` uses a pair of nested loops in combination with delta-coding in order to collapse 32,768 image rows to just 2.

Finally, a late change to the MNG specification was the addition of a new filter type, intrapixel differencing, borrowed from the LOCO-I compression method [14]. Unlike all of the other PNG filter types, which operate on any PNG image type and involve differences between bytes of *different* pixels, the new filter applies only to RGB or RGBA images, affects only the red and blue values by subtracting the green value from them, and feeds *that* result to the normal PNG filtering procedure. The combination of the new filter type and the existing PNG filters is collectively known as “filter method 64.” To the surprise of most PNG and MNG designers, this approach produces noticeably better compression than the standard PNG approach—on the order of 10% better for photographic images. If and when the PNG standard is updated incomparably, filter method 64 will probably be added to it, as well.

These five differences from PNG are immediately available to any MNG implementation. But in discussing deflate and the zlib compression format earlier, we also alluded to the fact that preset LZ77 dictionaries could, in principle, improve compression in a multi-image format. Although there has been no serious investigation of this possibility to date (and it may very well *not* prove useful in more than a tiny handful of situations), the basic idea would be to include a dictionary of commonly seen byte sequences in an as yet undesigned global MNG chunk—which may itself be compressed. All of the subsequent individual image streams would refer to this preset dictionary, which a decoder would pass to the zlib library using standard function calls. They would thereby reap immediate benefits in compression efficiency rather than slowly ramping up as the more common dynamic dictionary is generated. Such an approach could be quite beneficial to a collection of small images, all of which might be significantly smaller than the 32-KB deflate sliding window. On the other hand, the images would be completely indecipherable to any decoder that did not understand the hypothetical new preset-dictionary chunk—which is all of them, currently—so this idea is more of a theoretical possibility than a realistic compression approach.¹⁵

MNG’s last major difference from PNG is truly fundamental. Because lossy compression has always been recognized as far more efficient than lossless, at least for many image types, MNG developers decided to support standard JPEG compression as well as lossless PNG compression. For example, a photographic background (stored in JPEG format) can be combined with a lossless, transparent, animated text or graphic overlay (stored in PNG format), thus allowing the author to combine the best features of both formats within a single file. JPEG is also supported for

¹⁵ A similar but fully backward-compatible approach would be to concatenate all of the images into a single, large image and then use an appropriate MNG clipping region to select appropriate subframes. Of course, the applicability of either approach would depend entirely on the nature of the application(s) used to encode and decode the MNG file.

MNG alpha channels, which is not necessarily useful for the sharp-edged kind that are typical of antialiased images or text, but it can be highly effective for softer-edged alpha layers such as the feathered ovals often used in portraiture.

19.10 FURTHER READING

- Feldspar, A., An Explanation of the Deflate Algorithm. Available at <http://www.zlib.org/feldspar.html>.
- Gailly, J., comp.compression Newsgroup FAQs. Available at <http://www.faqs.org/faqs/by-newsgroup/comp/comp.compression.html>.
- Gailly, J., zlib home site. Available at <http://www.zlib.org/>.
- Miano, J., 1999. *Compressed Image File Formats*. Assoc. Comput. Mach., New York/Addison-Wesley Longman, Reading, MA.
- Murray, J. D., and W. vanRyper, 1996. *Encyclopedia of Graphics File Formats*, 2nd ed. O'Reilly and Associates, Sebastopol, CA.
- Nelson, M., and J. Gailly, 1996. *The Data Compression Book*, M&T Books, New York, NY.
- Roelofs, G., Multiple-Image Network Graphics (MNG) home site. Available at <http://www.libpng.org/pub/mng/>.
- Roelofs, G., 1999. *PNG: The Definitive Guide*, O'Reilly and Associates, Sebastopol, CA.
- Roelofs, G., Portable Network Graphics (PNG) home site. Available at <http://www.libpng.org/pub/png/>.

19.11 REFERENCES

1. Ziv, J., and A. Lempel, 1997. A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, Vol. IT-23, No. 3, [or Vol. IT-23(3)], pp. 337–343, May 1977. Available at <http://citeseer.nj.nec.com/context/8865/0>.
2. Ziv, J., and A. Lambel, 1978. Compression of individual sequences via variable rate coding, *IEEE Transactions on Information Theory*, Vol. IT-24, No. 5, pp. 530–536, September 1978. Available at <http://citeseer.nj.nec.com/context/4287/0>.
3. Welch, T. A., 1984. A technique for high performance data compression, *IEEE Computer*, Vol. 17, No. 6, pp. 8–19, June 1984. Available at <http://citeseer.nj.nec.com/context/4286/0>.
4. Boutell, T., G. Randers-Pehrson, *et al.*, PNG (Portable Network Graphics) Specification, Version 1.2. Available at <http://www.libpng.org/pub/png/spec/>.
5. Deutsch, L. P., DEFLATE Compressed Data Format Specification, Version 1.3. Available at <http://www.zlib.org/rfc-deflate.html>.
6. Deutsch, L. P., and J. Gailly, ZLIB Compressed Data Format Specification, Version 3.3. Available at <http://www.zlib.org/rfc-zlib.html>.
7. Fletcher, J., 1982. An arithmetic checksum for serial transmissions, *IEEE Transactions on Communication*, Vol. COM-30, No. 1, pp. 247–252, January 1982. Available at <http://citeseer.nj.nec.com/context/106076/0>.
8. Paeth, A. W., 1991. Image file compression made easy. In *Graphics Gems II* (James Arvo, ed.), Academic Press, San Diego, CA, 1991, ISBN 0-12-064480-0.
9. Schalnat, G. E., A. Dilger, G. Randers-Pehrson, *et al.*, PNG Reference Library (libpng). Available at <http://www.libpng.org/pub/png/libpng.html>.
10. Lehmann, A., G. Roelofs, and E. S. Raymond, gif2png. Available at <http://www.tuxedo.org/~esr/gif2png/>.
11. Lehmann, A., W. van Schaik, and G. Roelofs, pnmtopng. Available at <http://www.libpng.org/pub/png/apps/pnmtopng.html>.
12. Randers-Pehrson, G., pngcrush. Available at <http://pmt.sourceforge.net/pngcrush/>.
13. Randers-Pehrson, G., *et al.*, MNG (Multiple-image Network Graphics) Format, Version 1.0. Available at <http://www.libpng.org/pub/mng/spec/>.
14. Weinberger, M., G. Seroussi, and G. Sapiro, 1998. The LOCO-I Lossless Image Compression Algorithm: Principles and Standardization into JPEG-LS. Hewlett-Packard Laboratories Technical Report HPL-98-193R1, November 1998, revised October 1999. Also *IEEE Transactions on Image Processing*, Vol. 9, pp. 1309–1324, August 2000. Available at <http://www.hpl.hp.com/loco/>.

Facsimile Compression

KHALID SAYOOD

OVERVIEW

In this chapter we describe the compression schemes used for facsimile compression. We also briefly describe how these compression schemes have been incorporated into various international standards.

20.1 A BRIEF HISTORY

One of the earliest applications of lossless compression was to the compression of facsimile, perhaps because facsimile transmission is one of the oldest forms of digital transmission still in existence. The first facsimile machines were patented soon after Samuel Morse's patenting of the telegraph, in 1836. In 1843 A. Bain patented the *recording telegraph*, followed in 1850 by F. C. Blakewell, who patented the *copying telegraph*. The first commercial facsimile system was set up in France in 1865 using a system, called the pantelegraph, developed by G. Caselli.

It was almost a century before the precursor of the modern fax system made its arrival in 1968 with the development of the Group 1 facsimile standard. The Group 1 standard formalized in Recommendation T.2 of the CCITT, which is now ITU-T, coded the output of a photo-cell using two tones. In the CCITT recommendation a white pixel was represented by a tone at 1300 Hz and a black pixel was represented by a tone at 2300 Hz (in North America a white pixel was represented by a tone at 1500 Hz and a black pixel by a tone at 2300 or 2400 Hz). This standard allowed the transmission of an 8.5 in. by 11 in. page in 6 min. In 1976 the Group 1 fax standard was replaced by the Group 2 fax standard, which used vestigial sideband modulation to transmit a standard page in 3 min.

Compression came into the picture with the Group 3 and Group 4 standards which were published as ITU-T Recommendation T.4 and T.6. These standards permitted the transmission

THE SLEREXE COMPANY LIMITED

SAPORS LANE - BOOLE - DORSET - BH25 8ER
TELEPHONE BOOLE (945 13) 51617 - TELEX 123456

Our Ref. 350/PJC/EAC

18th January, 1972.

Dr. P.M. Cundall,
Mining Surveys Ltd.,
Holroyd Road,
Reading,
Berks.

Dear Pete,

Permit me to introduce you to the facility of facsimile transmission.

In facsimile a photocell is caused to perform a raster scan over the subject copy. The variations of print density on the document cause the photocell to generate an analogous electrical video signal. This signal is used to modulate a carrier, which is transmitted to a remote destination over a radio or cable communications link.

At the remote terminal, demodulation reconstructs the video signal, which is used to modulate the density of print produced by a printing device. This device is scanning in a raster scan synchronised with that at the transmitting terminal. As a result, a facsimile copy of the subject document is produced.

Probably you have uses for this facility in your organisation.

Yours sincerely,

Phil.

P.J. CROSS
Group Leader - Facsimile Research

Registered in England: No. 2088
Reg. Office: 101 Victoria Lane, Slough, Berks, England.

FIGURE 20.1

A test document image.

time for a page to come down to 1 min (assuming a 4800-bps modem, and excluding the time for handshaking). A "page" in this context refers to a white page with black pixels making up about 4% of the page as in the CCITT test chart 1, also known as the *slerexe page* shown in Fig. 20.1. The page is sampled at 1728 samples per line, with one page consisting of 2376 scan lines. Thus a standard page can be viewed as a bilevel image of size 2376 × 1728. If we were to try and transmit such a page without compression over a 4800-bps line, it would take a little more than 14 min. It would take almost an hour to send a four-page document. Clearly there is a need for compression.

The compression schemes used in Group 3 and Group 4 have also found an application as part of other *de facto* standards such as the Tagged Image File Format (TIFF), which in turn has given rise to a new Internet fax standard known as TIFF-FX. The increase in processing power available to devices has led to more efficient bilevel compression schemes. These are used in the JBIG and JBIG2 standards that were published as ITU-T Recommendation T.82 and T.88. Finally, with the ever-increasing popularity of color scanners and printers a new standard for color facsimile based on an approach called Mixed Raster Compression has been published by ITU-T in Recommendation T.44.

In this chapter we first describe the compression algorithms that are part of the various standards. We then briefly describe how these techniques have been incorporated in the international standards.

20.2 THE COMPRESSION ALGORITHMS

In this section we describe the various compression algorithms used for facsimile compression. These compression algorithms take advantage of the particular structure present in documents and are used in various international standards.

20.2.1 Modified Huffman

Looking at any document we can clearly see that black and white pixels are not randomly scattered through the page. They occur in groups. If we are looking at a single scan line, we will see a string of white pixels followed by a string of black pixels and so on. Therefore, it makes sense to encode the number of pixels in each string, or run, rather than the pixels themselves. This is called *run-length coding*. In order to obtain compression the run-lengths that occur more often should be encoded with shorter codewords than the run-lengths that occur less often. A coding scheme that does this is Huffman coding (see Chapter 3). However, the number of values that the run-lengths can take on is very large. You could have run-lengths all the way up to 1728. Creating a Huffman codebook of this size is clearly cumbersome. Furthermore decoding a Huffman code with 1728 entries in the codebook would impose an unrealistic computational load. The Modified Huffman (MH) algorithm represents the run-length r with two numbers, m and t , where

$$r = 64m + t.$$

Notice that t can take on values from 0 to 63, and m can take on values from 1 to 27. The MH algorithm uses separate variable-length codes to encode the values of m and t . The values of m are encoded using the *make-up* codes shown in Table 20.1 and Table 20.2. The values of t are encoded using the *terminating* codes shown in Table 20.3 and Table 20.4. Note that there are different codes for runs of white pixels and runs of black pixels. Table 20.5 lists the extended make-up codes for runs of black and white pixels.

The one-dimensional coding consists of codewords corresponding to alternate runs of white and black. We assume that the first run will be a run of white pixels. If this assumption is violated, then the code for a run-length of 0 white pixels is transmitted.

20.2.2 Modified READ

A facsimile is a two-dimensional object with a very high correlation between pixels on neighboring lines. The modified READ algorithm uses this fact to provide a substantial improvement in compression. The main feature of documents used in the two-dimensional algorithm is that the

Table 20.1 Make-up White Codes

Code	Length	Run	Code	Length	Run
11011	5	64	011010100	9	960
10010	5	128	011010101	9	1024
010111	6	192	011010110	9	1088
0110111	7	256	011010111	9	1152
00110110	8	320	011011000	9	1216
00110111	8	384	011011001	9	1280
01100100	8	448	011011010	9	1344
01100101	8	512	011011011	9	1408
01101000	8	576	010011000	9	1472
01100111	8	640	010011001	9	1536
011001100	9	704	010011010	9	1600
011001101	9	768	011000	6	1664
011010010	9	832	010011011	9	1728
011010011	9	896			

Table 20.2 Make-up Black Codes

Code	Length	Run	Code	Length	Run
0000001111	10	64	0000001110011	13	960
000011001000	12	128	0000001110100	13	1024
000011001001	12	192	0000001110101	13	1088
000001011011	12	256	0000001110110	13	1152
000000110011	12	320	0000001110111	13	1216
000000110100	12	384	0000001010010	13	1280
000000110101	12	448	0000001010011	13	1344
0000001101100	13	512	0000001010100	13	1408
0000001101101	13	576	0000001010101	13	1472
0000001001010	13	640	0000001011010	13	1536
0000001001011	13	704	0000001011011	13	1600
0000001001100	13	768	0000001100100	13	1664
0000001001101	13	832	0000001100101	13	1728
0000001110010	13	896			

transition from a white pixel to a black pixel, or from a black pixel to a white pixel, in any given line will occur at almost the same location as a similar transition in the previous line. The algorithm is a modification of a Japanese proposal to CCITT called the Relative Element Address Designate of READ algorithm [1]. Hence it is referred to as the modified READ or MR algorithm.

To see how the algorithm operates we use an example of two lines of a facsimile image shown in Fig. 20.2. Assume that the contents of the top line is known to both encoder and decoder. On the bottom line the last pixel known to both encoder and decoder is the pixel marked a_0 . We have labeled some of the transition pixels in the figure. The pixel labeled b_1 is the first pixel on the top line which is to the right of the pixel labeled a_0 and is of a different color. The pixel labeled b_2 is the first pixel to the right of the pixel labeled b_1 which is of a different color than pixel b_1 . The pixel labeled a_1 is the first pixel of a different color from the pixel labeled a_0 on the line being encoded and to the right of pixel a_0 . The pixel labeled a_2 is the first pixel to the right of pixel a_1 .

Table 20.3 Terminating White Codes

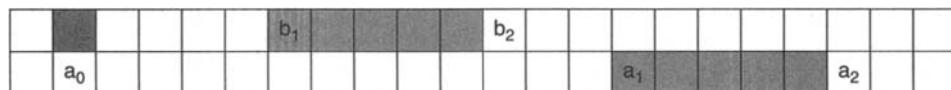
Code	Length	Run	Code	Length	Run	Code	Length	Run
00110101	8	0	0010111	7	21	00101011	8	42
000111	6	1	0000011	7	22	00101100	8	43
0111	4	2	0000100	7	23	00101101	8	44
1000	4	3	0101000	7	24	00000100	8	45
1011	4	4	0101011	7	25	00000101	8	46
1100	4	5	0010011	7	26	00001010	8	47
1110	4	6	0100100	7	27	00001011	8	48
1111	4	7	0011000	7	28	01010010	8	49
10011	5	8	00000010	8	29	01010011	8	50
10100	5	9	00000011	8	30	01010100	8	51
00111	5	10	00011010	8	31	01010101	8	52
01000	5	11	00011011	8	32	00100100	8	53
001000	6	12	00010010	8	33	00100101	8	54
000011	6	13	00010011	8	34	01011000	8	55
110100	6	14	00010100	8	35	01011001	8	56
110101	6	15	00010101	8	36	01011010	8	57
101010	6	16	00010110	8	37	01011011	8	58
101011	6	17	00010111	8	38	01001010	8	59
0100111	7	18	00101000	8	39	01001011	8	60
0001100	7	19	00101001	8	40	00110010	8	61
0001000	7	20	00101010	8	41	00110011	8	62
						00110100	8	63

Table 20.4 Terminating Black Codes

Code	Length	Run	Code	Length	Run	Code	Length	Run
0000110111	10	0	00000110111	11	22	000011011011	12	43
010	3	1	00000101000	11	23	000001010100	12	44
11	2	2	00000010111	11	24	000001010101	12	45
10	2	3	00000011000	11	25	000001010110	12	46
011	3	4	000011001010	12	26	000001010111	12	47
0011	4	5	000011001011	12	27	000001100100	12	48
0010	4	6	000011001100	12	28	000001100101	12	49
00011	5	7	000011001101	12	29	000001010010	12	50
000101	6	8	000001101000	12	30	000001010011	12	51
000100	6	9	000001101001	12	31	000000100100	12	52
0000100	7	10	000001101010	12	32	000000110111	12	53
0000101	7	11	000001101011	12	33	000000111000	12	54
0000111	7	12	000011010010	12	34	000000100111	12	55
00000100	8	13	000011010011	12	35	000000101000	12	56
00000111	8	14	000011010100	12	36	000001011000	12	57
000011000	9	15	000011010101	12	37	000001011001	12	58
0000010111	10	16	000011010110	12	38	000000101011	12	59
0000011000	10	17	0000011010111	12	39	000000101100	12	60
0000001000	10	18	0000001101100	12	40	000001011010	12	61
00001100111	11	19	000001101101	12	41	000001100110	12	62
00001101000	11	20	0000011011010	12	42	000001100111	12	63
00001101100	11	21						

Table 20.5 Extended Make-up Codes (Black and White)

Code	Length	Run
00000001000	11	1792
00000001100	11	1856
00000001101	11	1920
000000010010	12	1984
000000010011	12	2048
000000010100	12	2112
000000010101	12	2176
000000010110	12	2240
000000010111	12	2304
000000011100	12	2368
000000011101	12	2432
000000011110	12	2496
000000011111	12	2560

**FIGURE 20.2**

Two lines of a facsimile image.

which is of a different color from pixel a_1 . Note that the locations of pixel a_0 , pixel b_1 , and pixel b_2 are known to both encoder and decoder, while the locations of pixel a_1 and pixel a_2 are known only to the encoder.

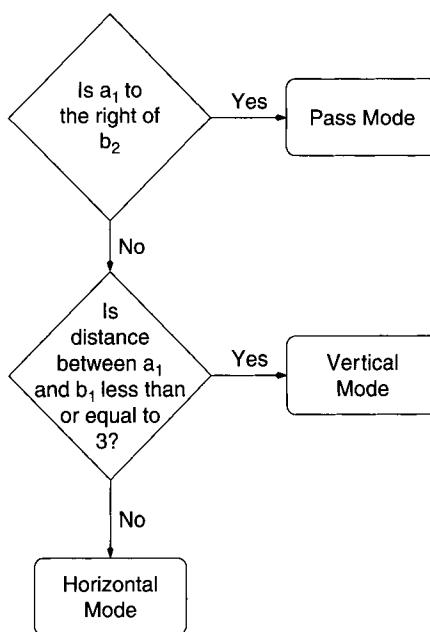
The encoder functions in one of three modes based on the relative locations of pixels a_1 , b_1 , and b_2 . If pixel a_1 is to the right of pixel b_2 , the encoder is in *pass mode*. If pixel a_1 is not to the right of pixel b_2 and within three pixels of pixel b_1 , the encoder is said to be in *vertical mode*. If neither of these conditions is satisfied, the encoder is said to be in *horizontal mode*. This procedure is summarized in Fig. 20.3.

In the *pass mode* the encoder sends the codeword 0001 to the decoder. This lets the decoder know that all pixels from pixel a_1 to just under pixel b_2 have the same color as pixel a_1 . The pixel under pixel b_2 becomes the new pixel a_0 and both encoder and decoder update the locations for the other transition pixels with reference to the new pixel a_0 . Encoding continues.

In the *vertical mode* the encoder encodes the location of a_1 using the following codes:

a_1 directly under b_1	1
a_1 to the right of b_1 by one pixel	011
a_1 to the right of b_1 by two pixels	000011
a_1 to the right of b_1 by three pixels	0000011
a_1 to the left of b_1 by one pixel	010
a_1 to the left of b_1 by two pixels	000010
a_1 to the left of b_1 by three pixels	0000010

Once the decoder learns the current location of the pixel a_1 , that pixel is relabeled a_0 and we repeat the procedure.

**FIGURE 20.3**

Selection of coding mode.

In the horizontal mode the encoder sends the code 001. The next two run-lengths, that is, the distance between the pixels labeled a_0 and a_1 and that between the pixels labeled a_1 and a_2 , are transmitted using modified Huffman codes.

Note that all the codewords mentioned above form a prefix code; no codeword is a prefix of any other codeword.

20.2.3 Context-Based Arithmetic Coding

Arithmetic coding was developed in its current form in the mid-1970s [2, 3] and has rapidly gained in popularity. The details of the arithmetic coding technique are presented in Chapter 5. Arithmetic coding is especially useful for the encoding of facsimile documents because of the structure of the data. The technique is particularly effective when the probabilities of the symbols being encoded are highly skewed. The probabilities of black and white pixels in facsimiles are highly skewed if we take into account the neighborhood, or the context, of the pixel being encoded. In context-based arithmetic coding the probabilities used to encode a particular symbol depend on the context in which that symbol occurs. A table of frequency counts is kept for the different context values. Consider the pixel marked X in Fig. 20.4. The bold outline delineates the context for the pixel X . As the pixels can take on only two values, the context, which contains 10 pixels, can take on 1024 values. In a context-based scheme one might have a separate table of frequency of occurrence for each context value.

Taking the contexts into account provides a much more accurate estimate of the probability of a given pixels, which in turn results in higher compression. The size and shape of the context are important factors in determining the effectiveness of this approach.

1	0	1	1	0	0	0	1	1	1	1	1	1	1
1	0	1	1	0	0	0	1	1	1	1	1	1	1
1	0	1	1	0	X	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?	?	?	?	?

FIGURE 20.4Context for coding pixel X .

20.2.4 Run-Length Color Encoding

This is a simple extension of standard run-length encoding and is formalized in ITU-T Recommendation T.45. The bitstream is divided into *header* and *data*. The header specifies the number of color components, the number of bytes used to represent each component value, and the number of different color values that are encoded.

The data consist of alternating run-lengths and values. The run-length is encoded using 1 or 3 bytes. If the run-length is between 1 and 255, a single byte is sufficient. The all-zero byte is used to signal the fact that the run-length is greater than 255. The next 2 bytes contain the value of the run. The color value is encoded using $n \times k$ bytes, where n is the number of components and k is the number of bytes used to represent the value of each component.

20.3 THE STANDARDS

The compression algorithms described in the previous section have been incorporated in a number of different international standards. These standards have evolved with the development of technology and the evolution of documents. In this section we briefly describe these standards.

20.3.1 ITU-T Group 3 (T.4)

The Group 3 standard contains two types of coding schemes; the one-dimensional coding scheme known as the Modified Huffman scheme and the two-dimensional scheme known as the Modified Modified READ scheme.

In the one dimensional scheme the modified Huffman codewords for each line of the facsimile is terminated by an end-of-line (EOL) codeword. This is a unique codeword consisting of 11 zeros followed by a one. The use of this symbol allows the decoder to synchronize after an error burst.

The two-dimensional coding is substantially more effective in terms of compression than the one-dimensional scheme. However, it is also vulnerable to error propagation. An error in one line will get propagated to the next line and so on. In order to prevent runaway error the Group 3 standard requires that every K lines one-dimensional coding be used where K is either 2 or 4.

The standard also includes provisions for optional error limiting and error correction modes. The error limiting mode is used only in the one-dimensional coding scheme. In this mode a line of 1728 pixels is divided into 12 groups of 144 pixels each. A 12-bit header is then constructed with 1 bit for each group of 144 pixels. If all pixels in a group are white, the corresponding bit in the header has a value of 0. If not, it has a value of 1. The all-white groups do not need to be encoded as the header contains all information necessary to reconstruct them. The other groups are encoded using modified Huffman codes.

In the error correction mode the bitstream is broken up into packets with each packet containing an error-detecting code. If a packet is detected to be in error, a request for retransmission is sent to the encoder. The request is not honored until the entire page has been transmitted. Each packet can be retransmitted a maximum of four times.

The end of document transmission is indicated by sending six consecutive EOLs.

20.3.2 Group 4 (T.6)

The Group 4 algorithms were standardized in ITU-T Recommendation T.6. Essentially, it takes advantage of the much cleaner digital lines to remove the restriction of using one-dimensional coding from the modified READ algorithm. As the two-dimensional algorithm is substantially more effective than the one-dimensional algorithm, this can dramatically increase the amount of compression. To reflect the fact that this algorithm is a (slightly) modified version of modified READ, it is called the *Modified Modified READ* algorithm or MMR.

20.3.3 JBIG and JBIG2 (T.82 and T.88)

In recent years two new standards have been developed by the Joint Bi-Level Experts Group, which is a joint committee of ITU-T, ISO, and IEC. These standards provide much higher compression as well as progressive transmission. The JBIG2 standard even allows for lossy compression of facsimile documents. The compression approach used in the JBIG standard is based on arithmetic coding. The standard describes a progressive transmission approach in which the context for encoding of a pixels can be made up of pixels from the neighbors of the pixel in the current layer or a lower resolution layer.

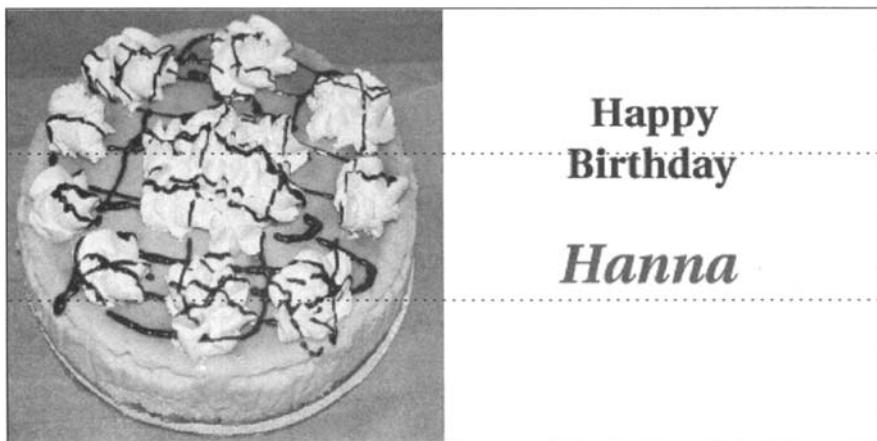
The more recent JBIG2 standard reflects the evolving nature of documents that increasingly contain images. The document to be encoded is segmented into *halftone regions*, *symbol regions*, and *generic regions*. Halftone regions are parts of the document containing halftone images, symbol regions are parts of the document containing text symbols, and the generic regions are regions that do not fit into either of the first two categories. Each region is encoded using techniques appropriate to the type of structures present in the data corresponding to the region.

Details on both the JBIG and the JBIG2 standards can be found in Chapter 17.

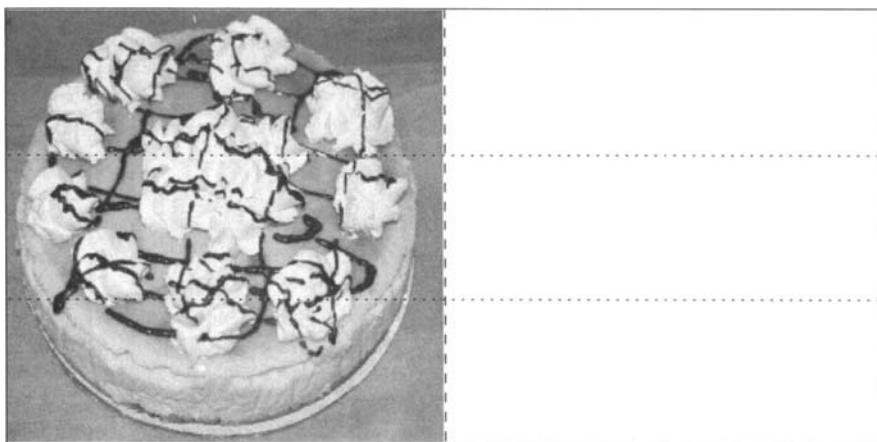
20.3.4 MRC—T.44

As society develops and evolves so do documents. In recent years, documents contain more and more multicolored text and images. When such documents are converted to bilevel images, there can be considerable loss of resolution. This level of quality can be unacceptable to consumers. To deal with the changing nature of the document and consumer expectations the ITU-T has accepted the Mixed Raster Content (MRC) approach as a new standard for facsimile transmission.

The MRC approach recognizes the fact that no compression scheme works well with both continuous tone images and text. The JPEG standard and the new JPEG2000 standard provide high compression with excellent quality for natural images. However, if the text portion of the document is compressed to the same level, there is a loss of crispness in text. The JBIG and JBIG2 standards provide excellent compression for text but they do not provide a very high level of compression for continuous tone images. Recognizing this fact the MRC strategy is to decompose the facsimile image into three layers (there is an option to increase the number of layers). The *background layer* consists of the portions of the document that consist of continuous tone images. The foreground layer consists of portions of the image containing text or line art.

**FIGURE 20.5**

Test image. (See also color insert.)

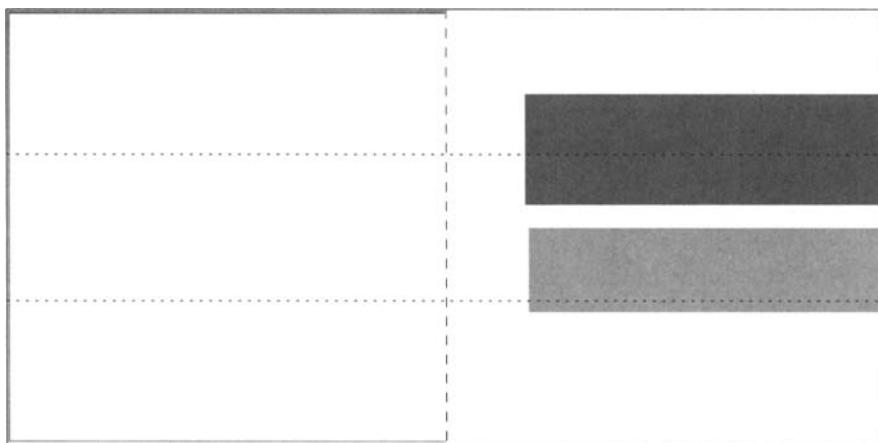
**FIGURE 20.6**

Background for the test image. (See also color insert.)

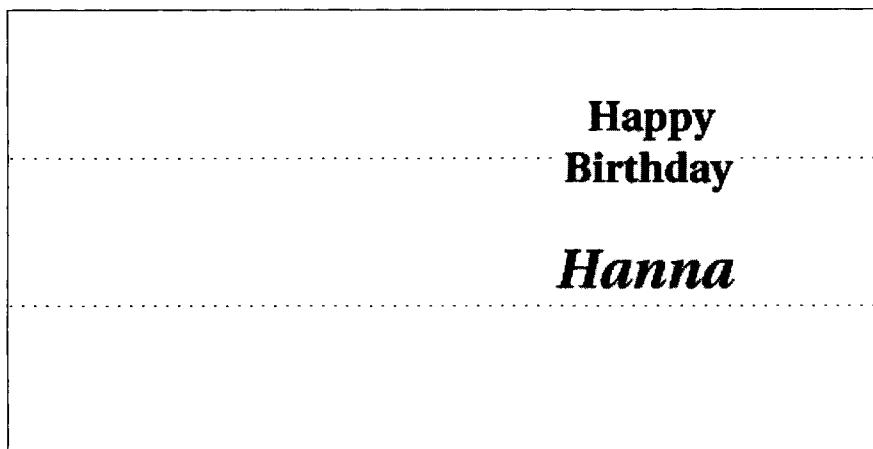
A bilevel mask layer provides directions for the reconstruction of the image. A black pixel (value of 1) indicates that that particular pixel in the reconstructed image should come from the foreground layer and a white pixel indicates that the pixel in that location in the reconstructed image should come from the background layer.

As an example consider the document shown in Fig. 20.5 (see also color insert). The left half of the picture contains a continuous tone image, while the right half of the image contains multicolored text (we will describe the meaning of the dashed and dotted lines in this and the following figures later in the chapter). The background layer, foreground, and mask layers for this document are shown in Fig. 20.6–20.8 (see also color insert for Figs. 20.6 and 20.7).

Because of memory restrictions in the coder, the document can be partitioned into stripes. As an example we have partitioned the test document shown into three stripes. Notice that not all stripes in Fig. 20.5 contain both foreground and background layers. Furthermore notice that in the

**FIGURE 20.7**

Foreground for the test image. (See also color insert.)

**FIGURE 20.8**

Mask for the test image.

foreground and background not all of the stripe is occupied by picture or text. To take advantage of these the recommendation allows three kinds of stripes: three-layer stripes (3LS), two-layer stripes (2LS), and one-layer stripes (1LS). The three-layer stripes contain mask foreground and background layers. The mask layer is transmitted first followed by the background and then the foreground layer. The two-layer stripes contain the mask layer and either the foreground or the background layer. The layer not included is set to a constant value. This type of stripe is useful when the stripe contains monochrome text or line art and a continuous tone image or no image and multicolored text. In this case also the mask layer is transmitted first followed by the foreground or background layer. The one-layer stripe is useful when the stripe contains only a continuous tone image or monochrome text or line art. This is the case for the lowest stripe in the test image.

Only the mask layer is required to span the entire width of the stripe. The other layers can be specified to cover only part of the width of the strip. In the example shown in the figures, in the background stripe the data to the right of the dashed line need not be encoded.

The mask layer can be coded using Modified Huffman (T.4), Modified READ (T.4), Modified Modified READ (T.6), or JBIG (T.82). The coder used is specified in the set of bytes starting with the 13th byte in the Start of Page marker segment. Currently with only these four coding schemes the coder can be specified by setting one of the bits in a single byte. If the number of encoders gets to be greater than or equal to 7, there will be a need for more than a single byte. The other two layers can be encoded using either JPEG (T.81) or JBIG (T.82). The encoding method used is specified in the byte(s) following the bytes that specify the coder for the mask layer in the Start of Page marker segment.

Unlike previous standards there is an inherent assumption in T.44 and the MRC approach that there will be new developments in compression and in the nature of documents. This flexibility should make it a valuable standard for years to come.

20.3.5 Other Standards

There are a number of other facsimile standards including military and NATO (STANAG) standards. However, they are principally variations of the standards described above. With the increased use of the Internet for a variety of communication activities previously restricted to other channels, the Internet Engineering Task Force has been working on a standard for Internet fax. The emerging standard is popularly known as TIFF-FX. This standard allows for inserting data encoded using the facsimile standards described above into the tagged image file format. The TIFF-FX standard can be viewed as a superset of all the standards described in this chapter. Its various profiles support T.4, T.6, T.44, and T.82 as well as JPEG (T.81).

20.4 FURTHER READING

- The JBIG and JBIG2 standards are described in Chapter 17.
- Huffman coding and arithmetic coding are described in Chapters 4 and 5, respectively.
- The various ITU standards themselves are quite readable and are the ultimate source for information.

20.5 REFERENCES

1. Yasuda, Y., 1980. Overview of digital facsimile coding techniques in Japan. *IEEE Proceedings*, Vol. 68, pp. 830–845, July 1980.
2. Pasco, R., 1976. *Source Coding Algorithms for Fast Data Compression*, Ph.D. thesis, Stanford University.
3. Rissanen, J. J., 1976. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, Vol. 20, pp. 198–203, May 1976.

PART V

Hardware

This Page Intentionally Left Blank

Hardware Implementation of Data Compression

SANJUKTA BHANJA
N. RANGANATHAN

21.1 INTRODUCTION

With the advancement of information technology, large-scale information transfer by remote computing and the development of massive information storage and retrieval systems have witnessed a tremendous growth. The growth of these systems implies the need for efficient mechanisms for storage and transfer of enormous volumes of data. Data compression is the reduction of redundancy in data representation in order to decrease data storage requirements and data transfer costs. It is the process of transforming a body of data into a smaller form from which the original or some approximation of the original can be recovered at a later time. The compression methods used depend on the type of data such as text, image, video, graphics, and audio, which collectively represent the multimedia information. In lossless data compression, the data that are compressed and subsequently decompressed must always be identical to the original data. In lossy compression, the decompressed data are an approximation of the original data. Lossless methods are referred to as entropy coding methods and are used in the compression of text and certain types of images; lossy methods are used in compressing image, video, and audio data that contain significant redundancy. Furthermore, most hardware products for image and video compression typically combine lossless and lossy compression capabilities together in the same product. For example, in the JPEG image compression implementations, the lossy compression includes hardware implementation of lossless coding, making it difficult to separate them. Thus, in this chapter, we include discussions on hardware implementation of both lossless and lossy compression.

In recent years, the demand for data compression and the need to develop efficient compression algorithms and implementations have increased considerably due to the increased use of data compression within commercial, scientific, and statistical information systems, the World Wide Web,

document delivery systems, and communications networks. Furthermore, in the field of multimedia, high-speed data compression has become critical to many communication and entertainment applications such as digital TV, video phone, video-on-demand, Internet, DVD, interactive games, and multimedia education. It is pointed out in [9] that the advances seen in multimedia and the proliferation of multimedia applications are due mainly to the progress in three areas: standards, networking and VLSI.

During the 1980s, data compression was primarily used through software implementations, and the time and space requirements of the compression and decompression routines made it less attractive. The emergence of compression standards in the early 1990s and the growth in VLSI technology made efficient hardware implementations of data compression a viable alternative. For example, as soon as the baseline JPEG standard was established, several companies such as LSI Logic, C-Cube, and Intel announced single-chip and multichip VLSI systems implementing the JPEG. Today, data compression hardware has become a critical component in virtually every household, either within the cable TV switch box or in a high-speed modem connecting the personal computer to the Internet. The main goal of compression hardware systems is to achieve high performance and throughput at low cost and with low power consumption. Although compression is often implemented using programmable processor cores such as DSP or multimedia processors, the design of dedicated architectures became essential to meet the performance requirements. The establishment of standards made it possible to design and develop special-purpose architectures and application-specific VLSI systems to implement sophisticated compression techniques that can process in real time. In fact, when the various algorithms and methods are considered for inclusion in the standards, one of the major criteria is their amenability to hardware implementation.

In developing high-speed VLSI architectures, several attributes are important: pipelining and parallel processing, high-speed memory access, scalability with technology and size, programmability, low power consumption, and low cost. The compression algorithms in general are computationally intensive in nature. Thus, it is important to exploit all inherent parallelism in the algorithms and map them onto parallel architectures. The use of array processors (for exploiting spatial parallelism) with nearest-neighbor communication and extensive pipelining of the various logic modules within the processors as well as the entire system architecture (for exploiting temporal parallelism) is critical to obtaining high throughput. Numerous systolic, SIMD, and MIMD array architectures have been proposed in the literature for data compression and decompression. The design of efficient memory systems that can store and provide image or video datastreams to the hardware processors keeping up with the high processing rate is critical to the performance of the system.

The architecture as well as the implementation must be highly scalable with technology. The scalability of an architecture determines the life span of the architecture in the market. It indicates whether an architecture can support further developments in terms of VLSI technology, data compression algorithms, and the emerging new standards. The advances in VLSI technology yielding lower gate delays, higher packing density, faster clocks, etc., directly impact the architecture. The minimum feature size (the size of the smallest transistor or the minimum wire width possible) keeps decreasing, which in turn could significantly increase the integration capability (the maximum number of devices that can be packed within a single die or chip). For example, a hardware system designed using $0.5\text{-}\mu\text{m}$ technology (the minimum feature size being $0.8\text{ }\mu\text{m}$) must be easily portable into a $0.18\text{-}\mu\text{m}$ design when needed. Conversely, one may want to produce a low-cost system for certain applications by using an older, cheaper technology. The process of scaling up or scaling down in VLSI technology can have different implications in terms of the performance of the overall system. Thus, the architecture must be easily amenable to the changing technology. The size scalability corresponds to the architecture and the implementation being able to adapt to an increase or a decrease in the size of any system parameter or configuration. For example, in a

networked multimedia system, such changing parameters could be the video frame size, the frame rate, network traffic, performance requirements, quality of service parameters, and application behavior.

Programmability is another important criterion in multimedia hardware systems so that they can handle different standards and also variations in standards or algorithms that occur in time. However, while programmable processors are flexible, they are slower than dedicated hardware in general. Often, the computation-intensive tasks such as DCT (Discrete Cosine Transform) coding or variable-length encoding, which are common to most image and video compression standards, are implemented as dedicated special-purpose hardware to gain maximum speed. The optimization of power consumption has become important in the context of battery-operated systems as in wireless and mobile computing environments. The reduction of power consumption is targeted at almost every level: algorithmic, architectural, gate, circuit, and device levels. Thus, besides low cost and high performance, low power has become an important design goal in wireless multimedia systems.

Numerous architectures as well as commercial VLSI products for data compression have come into existence over the past decade. There are a plethora of publications in the area of hardware data compression that can be found in the literature. It is almost impractical to cite and cover every important contribution within a single chapter. Thus, our intention in this chapter is to provide a brief overview that can serve as a starting point. We include brief case studies of a few architectures merely to serve as examples for reading. The interested reader is referred to the vast literature for further exploration. The rest of the chapter is divided into sections that discuss text, image, video, and commercial products.

21.2 TEXT COMPRESSION HARDWARE

A wide variety of lossless compression methods have been proposed in the literature for text data. Examples include Huffman coding [21], the multigroup technique [5], run-length coding [77], Lempel-Ziv coding [43], arithmetic coding [27], and other dictionary-based methods. Many hardware architectures for text compression implement tree-based codes, also referred to as variable-length encoders. Other popular compression methods implemented in hardware are arithmetic coding, Lempel-Ziv coding, and some dictionary-based methods.

The implementation of variable-length codes is difficult due to the fact that the codeword boundary cannot be determined until the previous codeword has been decoded completely, indicating a bit-sequential nature of processing. This in turn puts limits on the decoding speed and throughput. A method to address this problem was reported in [5, 6] using the concept of a reverse binary tree for static Huffman encoding and decoding. Also, an adaptive and pipelined design for generating the Huffman tree codes given the statistical distribution of characters in a text file was given. A high-speed entropy decoder for VLE codes based on programmable logic arrays was proposed in [61]. A memory-based architecture called MARVLE was proposed in [4] for implementing tree-based codes in which the nodes of the tree representing the codes are mapped to specific memory locations based on a specific algorithm that aids in fast encoding and decoding of text files using tree-based codes. Area-efficient architectures with memory-mapping schemes were proposed in [29]. Hardware designs using content addressable memory for implementing dynamic Huffman coding can be found in [52].

The principle behind the Lempel-Ziv (LZ) methods is to find the longest strings that match a previous occurrence of that string within the same file or a statically predefined or a dynamically constructed dictionary of strings and then replace the current occurrence with a pointer to the previous occurrence or to the location within the dictionary. The LZ methods and their variations

have been excellent candidates for VLSI implementation. Several systolic VLSI designs for implementing the Lempel–Ziv methods can be found in the literature [10, 11, 44, 56, 71, 73, 75]. Furthermore, CAM-based designs can be found in [46] and other custom architectures can be found in [20, 90]. Since the systolic designs are ideal architectural candidates for VLSI implementation, we will describe one such approach [64] as an example later in this section.

A massively parallel architecture for text compression using textual substitution was introduced in [44]. A new algorithm for text compression based on textual substitution and its implementation as an ASIC using systolic arrays was proposed in [25]. Hardware implementations of arithmetic coding can be found in [32, 45]. The work in [45] describes a lossless compression method based on arithmetic coding that uses fuzzy inferencing and its hardware implementation. The Rice algorithm [72] for lossless compression and its implementation as a VLSI chip set are described in [42]. More architectures can be found in the literature, not covered here for want of space. In the rest of this section, we briefly discuss two architectures, one for tree-based codes [4] and another for LZ coding [64], as examples.

21.2.1 Tree-Based Encoder Example

Tree-based codes are commonly used for text compression in which the codes can be represented by trees, such as binary trees or quad trees, where the sequence of 0's and 1's on the unique path from the root of the tree to a leaf node represents the code for the symbol represented by the leaf node. Some examples of these codes are the Shannon–Fano code, the Huffman code, the Elias code, and the Fibonacci code. These codes are derived based on the statistical distribution of the various characters within a file in text compression. The tree-based codes are variable-length codes in which compression is achieved by assigning short codewords to high-probability symbols and assigning long codewords to low probability symbols. In order for these codes to be uniquely decipherable, they must also have the prefix property, which states that no code can be a prefix of any other code.

An example of a Huffman tree for the symbols a, b, c, d, e, f, and g with probabilities of 0.2, 0.15, 0.10, 0.25, 0.05, and 0.15 is given in Fig. 21.1a. In order to facilitate the implementation of the compression process in hardware, a reverse binary tree is derived for the tree representing the code. A reverse binary tree [6] is a labeled binary tree whose leaf nodes and some of the internal nodes represent the symbols to be coded. The sequence of 0's and 1's on the unique path from the symbol node to the root of the tree represents the code for that symbol. The reverse binary

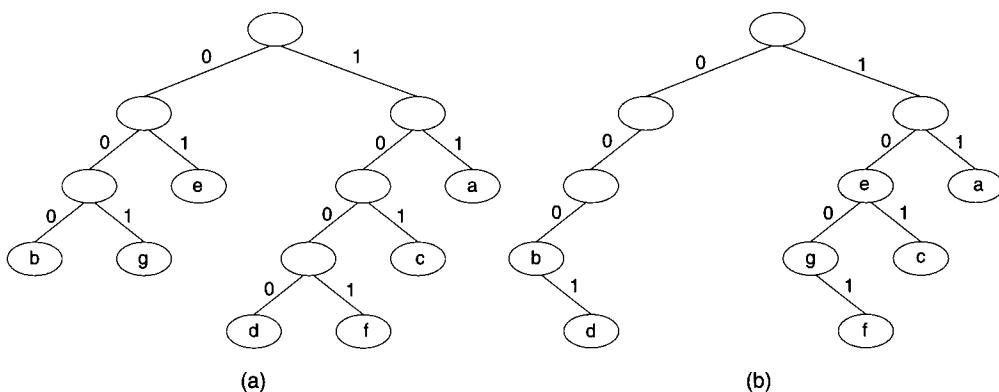
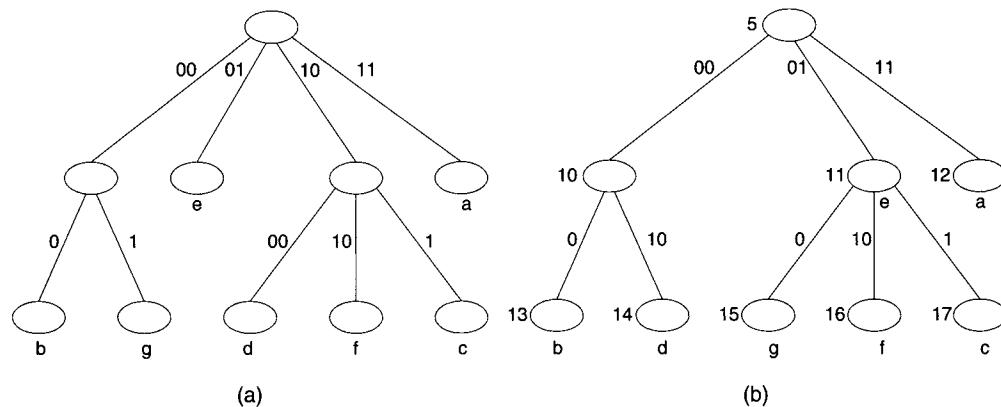


FIGURE 21.1

Huffman and reverse tree example.

**FIGURE 21.2**

Two-bit trees for Huffman and reverse tree examples.

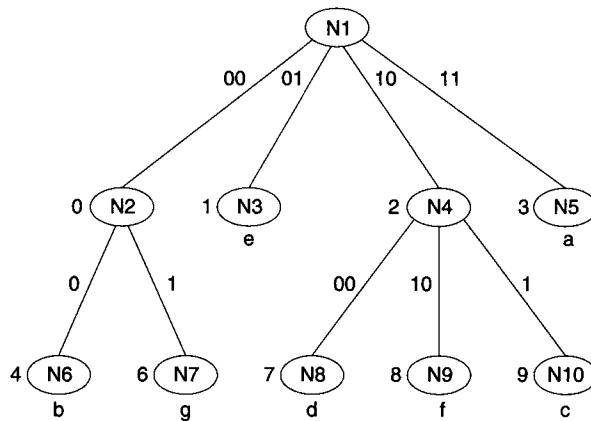
tree for a code can be constructed by using the reverse codes for each symbol. The reverse binary tree for the Huffman tree in Fig. 21.1a is given in Fig. 21.1b.

The encoding and decoding are inherently serial operations since the tree must be traversed one edge at a time. In order to speed up the process, a multibit tree may be used. In a multibit tree, each edge of the tree represents a maximum of k bits of the input code. If the length of a codeword is n bits, then it is represented by n/k edges from the root of the tree to a leaf node of which only the last edge (the one terminating at the leaf node) could possibly have a label less than k bits long. The k -bit reverse tree is analogous to the 1-bit reverse tree where the codes are reversed k bits at a time. Each node of the k -bit tree can have a maximum of $2k$ children. The 2-bit trees corresponding to the Huffman code tree and the reverse tree in the above example are shown in Fig. 21.2a and 21.2b. The architecture of MARVLE is centered around a memory device in which the code trees are mapped and stored for encoding and decoding.

21.2.1.1 Memory Mapping

A memory map of a tree is defined by associating a distinct memory location with each node of the tree [4]. The mapping of the decoding tree is relatively more complex than the mapping of the encoding tree. The following procedure can be used for the mapping of the decoding tree: Given a k -bit decoding tree with n nodes, of which p nodes have at least two child nodes and the remaining (np) nodes are either leaf nodes or nodes with a single child, the number of child nodes of a node N_i is C_i , where $1 \leq C_i \leq 2^k$. The memory mapping proceeds as follows:

1. Each edge that connects a node N_i and a child node N_j has a label $L_{ij} = x_1, x_2, \dots, x_s$, where $s \leq k$ and each x_i is a single bit: 0 or 1.
2. An integer B_{ij} is assigned to each L_{ij} such that $B_{ij} = \sum_{i=1}^s 2^{k-i} x_i$.
3. For each node N_i a corresponding positive integer M_i will be assigned and a set of memory addresses will be defined for each node N_i as $\text{Mem}(N_i) = \{M_i + B_{ij}\}, j = 1, 2, \dots, C_i$.
4. Steps 1–3 are repeated for each N_i , $1 \leq i \leq p$.
5. For each node N_i , a value for the corresponding M_i is chosen such that all $\text{Mem}(N_i)$ are distinct.
6. Map the remaining nodes N_{p+1}, \dots, N_n to distinct positive integers outside the memory map.

**FIGURE 21.3**

Memory map for Huffman example.

The mapping process will be illustrated using a 2-bit Huffman tree where each edge of the tree represents a maximum of 2 bits of the code. The 2-bit Huffman tree for the example in Fig. 21.1a with the nodes labeled N1 through N10 is shown in Fig. 21.3.

The equations for $\text{Mem}(N_i)$ are as follows:

$$\begin{aligned}\text{Mem}(N1) &= \{M1 + 0, M1 + 1, M1 + 2, M1 + 3\} \\ \text{Mem}(N2) &= \{M2 + 0, M2 + 2\} \\ \text{Mem}(N3) &= \{M3 + 0, M3 + 1, M3 + 2\}.\end{aligned}$$

The assignment of $M_1 = 0$, $M_2 = 5$, and $M_3 = 8$ yields a valid memory map. This maps the nodes N_2 through N_{10} to memory addresses 0, 2, 1, 3, 5, 7, 8, 9, and 10, respectively. Note that the mapping has not utilized the addresses 4 and 6. The number of memory locations not utilized in the block of contiguous memory addresses for the decoding tree will be called the gap W . For this example, $W = 2$. The encoding map is created using the reverse binary tree by mapping each node of the tree to a distinct memory location. The choice of memory locations is simplified by the fact that the algorithm starts at a symbol node and traverses up the tree until the root node is reached. Since each node has only one parent, the addresses can be arbitrarily chosen as long as they are distinct. One possible memory map for the encoding tree is shown in Fig. 21.3 with memory addresses shown adjacent to each node.

The memory required for implementing the mapping in the MARVLE architecture is composed of 12-bit data words. The 12-bit words store data and control bits related to the nodes and edges of the encoding and decoding trees. Each memory word contains a 9-bit field for the data word and three control bits, T, S1, and S2. The various fields of the memory word have different meanings for the encoding and decoding memory maps. The encoding tree contains three types of nodes that need to be mapped onto the memory locations: (i) regular nodes, (ii) special nodes, and (iii) terminal nodes. The regular node is a non-leaf node in which all the edges from the node to its children have labels of 2 bits. The special node is a non-leaf node in which at least one of the edges connecting the node to its children has a single bit label. The terminal node is a leaf node that represents an output symbol. The data field for a non-terminal node (regular or special) contains the value of M_i for the node, and the data field for a terminal node contains a binary representation of the symbol at that node (e.g., ASCII). The three control bits T, S1, and S2 are used to identify the different types of nodes.

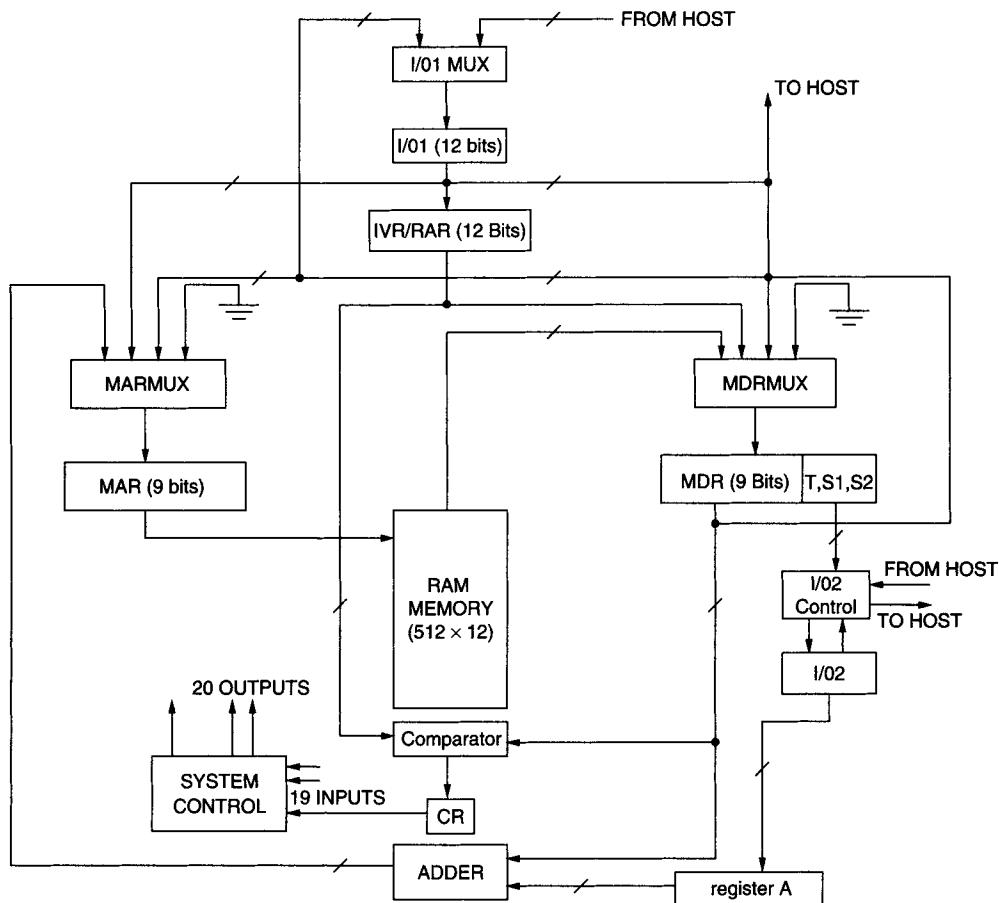


FIGURE 21.4
MARVLE architecture.

21.2.1.2 MARVLE Architecture

The main components of the MARVLE architecture are: (i) a 512×12 bit random access memory (MEM) consisting of 12-bit words having the format described in the previous section, (ii) a 9-bit memory address register (MAR), (iii) a 12-bit memory data register (MDR), (iv) a 12-bit parallel interface register (I/O_1), (v) a 2-bit parallel interface register (I/O_2), (vi) a 2-bit register (A) for compiling the offset value during the decoding process, and (vii) a 12-bit initial value register/root address register (IVR/RAR) that is used to hold the root address of the encoding tree and the initial T,S1, and S2 values for the root of the decoding tree. The notation $\text{MEM}[\text{ADDRESS}]$ denotes the data stored in the memory at the location ADDRESS.

A block diagram of the architecture is shown in Fig. 21.4. The major elements of the architecture are the system memory (MEM) and a set of registers. The registers are divided into two groups internal registers and input/output (I/O) registers. The internal registers are the MAR, MDR, IVR/RAR, and A and the comparator register that latches the output of an equality comparator. The compression and decompression algorithms require two other functional units, an adder and a bit-wise equality comparator. This adder is used during the decoding process to add the 2-bit offset in the A register to the base address contained in the MDR. This is a special-purpose adder that adds 2 bits to 9 bits and produces a 9-bit result. No attempt is made to deal with overflow.

since in the correct operation of the chip this situation never occurs. The bit-wise equality comparator is used during the encoding process to compare the 9 high-order bits of the IVR/RAR and the 9 high-order bits of the MDR to determine when the root of the tree is reached. The basic steps of the encoding algorithm are as follows:

1. Load the address of the symbol to be encoded into the MAR.
2. Check to see if the address is the root address of the tree, indicating that the encoding is done. If the address is the root address of the tree, go to step (1) (or stop if there are no more symbols).
3. Access the memory and place the contents of the location pointed to by the MAR in the MDR.
4. Output the bit(s) of the encoded symbol to the host. Output S_1 if $T = 0$ or S_1 and S_2 , if $T = 1$.
5. Load the MAR with the 9 high-order bits of the MDR (address of parent of the current node). Go to Step (2).

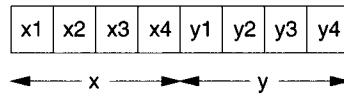
The basic steps of the decoding algorithm are as follows:

1. Load the MDR with the initial values for the root of the decoding tree.
2. If $T = 0$, output the decoded symbol to the host and go to (1).
3. Place the next bit on the input bitstream in A[2]. This is the high-order bit of the offset to the next node.
4. Determine the low-order bit of the offset and place it in A[1]. The low-order bit of the offset is determined as follows:
 If $S_1 = S_2 = 0$, $A[1] =$ the next bit on the input bitstream.
 If $S_1 = S_2 = 1$, $A[1] = 0$.
 If $(S_1 \neq S_2)$ and $(A[2] = S_2)$, $A[1] = 0$
 If $(S_1 \neq S_2)$ and $(A[2] \neq S_2)$, $A[1] =$ the next bit on the input bitstream.
5. Add the offset in A to MDR[12..4] and place the result in the MAR.
6. Access the memory and place the contents of location pointed to by the MAR in the MDR. Go to Step (2).

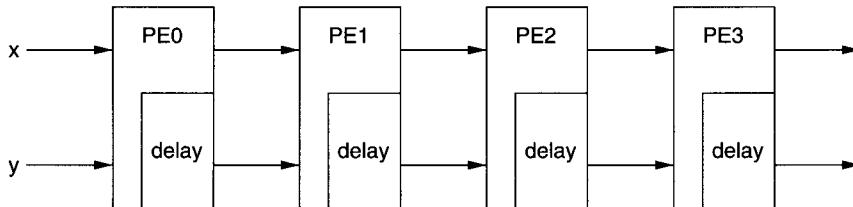
A VLSI chip prototype of the above architecture was reported to yield a compression rate of 95.2 Mbits/s and a decompression rate of 60.6 Mbits/s operating at 83.3 MHz [4]. In order to improve the performance it was pointed out that the architecture can be expanded to handle higher order trees such as 3- or 4-bit trees. This would speed the process because the limiting factor in performance is the memory cycle, which limits the clock speed of the chip. Increasing the dimension of the coding and decoding trees would mean that more bits will be decoded on each cycle of the algorithm, so that the rates would increase proportionally. However, increasing the tree size would increase the complexity of the control logic and the memory-mapping procedure. Hence, there is a trade-off between the increase in the tree dimension and the complexity of control and decoding time.

21.2.2 Lempel-Ziv Encoder Example

In this subsection, we describe a simple hardware implementation of the basic LZ77 compression technique that appeared in [64]. The LZ technique lends itself well to a systolic parallel array implementation. The systolic algorithm is explained using an example with a buffer of size 8 shown in Fig. 21.5. The first four locations of the buffer, labeled x_1 through x_4 , contain symbols

**FIGURE 21.5**

Buffer of size 8.

**FIGURE 21.6**

A systolic array of four processors (PEs).

that have already been processed and the other four locations, labeled y_1 through y_4 , contain symbols to be processed. The symbols are input to the buffer from the right and are shifted out from the left.

The following four sets of comparisons must be done in sequence in order to find the maximum matching substring, as can be seen from the pseudo-code given above:

$$\begin{array}{lll} x_1-y_1 & x_2-y_2 & x_3-y_3 \\ x_2-y_1 & x_3-y_2 & x_4-y_3 \\ x_3-y_1 & x_4-y_2 & y_1-y_3 \\ x_4-y_1 & y_1-y_2 & y_2-y_3 \end{array}$$

Here a dash indicates an equality comparison between the symbols. The set with the maximum number of matches in sequence determines the required substring. “In sequence” indicates that if any of the comparisons in a set fails, the succeeding comparisons in that set do not count as successful ones. For example, if x_1-y_1 was a successful comparison and if x_2-y_2 was not a successful comparison, then the result of the x_3-y_3 comparison is ignored and the match length is 1 in that case. The number of comparisons in the sequential algorithm is $O(n^2)$. The above set of comparisons can be reorganized in the following fashion:

$$\begin{array}{llll} x_1-y_1 & x_2-y_1 & x_3-y_1 & x_4-y_1 \\ x_2-y_2 & x_3-y_2 & x_4-y_2 & y_1-y_2 \\ x_3-y_3 & x_4-y_3 & y_1-y_3 & y_2-y_3 \end{array}$$

Let us consider four processors operating in parallel, each performing one vertical set of comparisons. Each processor would require $n - 1$ time units ($n - 1$ time units in general) to complete its set of comparisons. This results in the reduction of comparison time to $O(n)$, which is achieved by performing n comparisons in parallel. Consider the systolic array of four processing elements (PEs) as shown in Fig. 21.6. The “delay” block in each PE delays the y values by one time step. A space–time diagram illustrating the sequence of comparisons as performed by each PE is given in Fig. 21.7. The data brought into PE0 are routed systolically through each processor from left to right. In Fig. 21.7, the first three comparisons shown in each column indicate the comparisons to be performed. The symbol z denotes a dummy symbol and the comparisons involving z are unused. The sequence of comparisons to be performed by the PEs is explained as follows:

cycle	PE0	PE1	PE2	PE3
1	x_1-y_1			
2	x_2-y_2			
3	x_3-y_3	x_2-y_1		
4	x_4-z lgt-max	x_3-y_2		
5	y_1-z	x_4-y_3	x_3-y_1	
6	y_2-z	y_1-z lgt-max	x_4-y_2	
7		y_2-z	y_1-y_3	x_4-y_1
8			y_2-z lgt-max	y_1-y_2
9				y_2-y_3
10				lgt-max

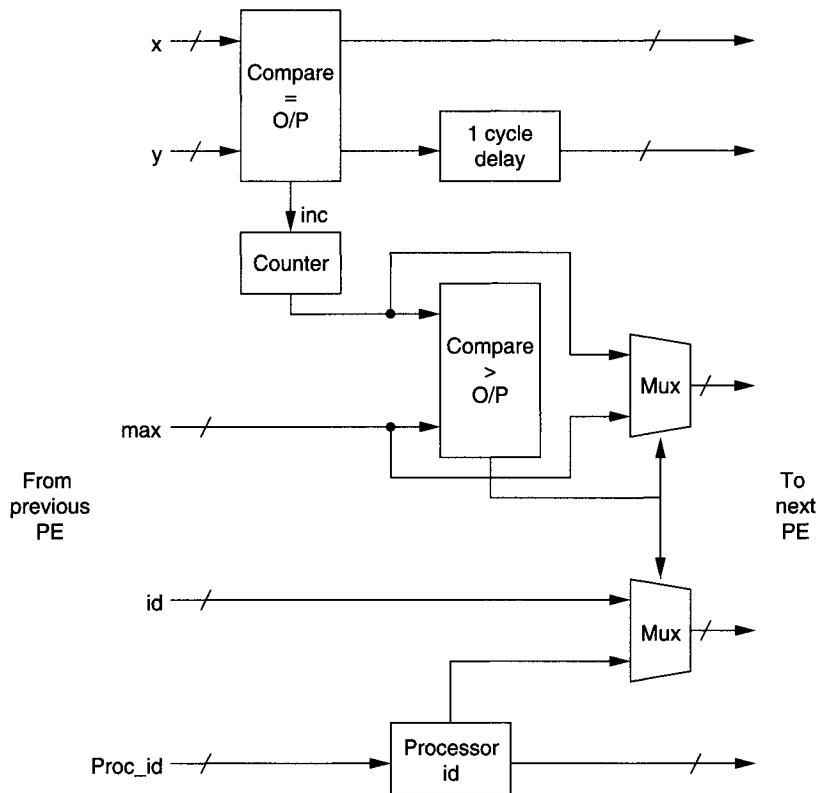
FIGURE 21.7

The space–time diagram for systolic data flow.

The data are input to PE0, which is passed systolically to the other PEs. In the first time unit, x_1 and y_1 are compared at PE0. In the second time unit, x_2 and y_2 are compared and x_1 flows to PE1 and y_1 is delayed by one cycle. In the third time unit, x_3 and y_3 are compared at PE0. At this time, y_1 gets to PE1 along with x_2 and PE1 performs its first comparison. After the third cycle, PE0 completes all its required comparisons and stores an integer specifying the number of successful comparisons in a register called length. Another register, max, holds the maximum matching length obtained from the previous PEs. In the fourth time unit, PE0 compares the values of max (which for PE0 is 0) and length and the greater of the two is sent to the max register of the next PE. Although x_4 is not used by PE0, it is used by the following PEs. Similarly, y_1 and y_2 are input to PE0 in the following time units as seen in the space–time diagram. The comparisons involving “z” are ignored. The result of the *length–max* comparison is sent to the next PE after a delay of 1 time unit for proper synchronization. Finally, the max value emerging from the last PE (PE3 in this case) is the length of the longest matching substring. There is another register called id, whose contents are passed along with the max value to the next PE. Its contents indicate the id of the processor where the max value occurred, which becomes the pointer to the match. The method takes $3n$ units in general to find the maximum possible match length, where n is the number of PEs.

21.2.2.1 The Processing Element

Each PE performs the following set of functions: (i) The incoming symbols are compared to check whether they are equal. If they are equal, a counter is incremented each time until an unsuccessful comparison occurs. The output of the counter gives the length of the match. (ii) Once a PE has found its match, it compares the match length with the max from the previous PE. The greater of the two is output as the new max to the next PE. (iii) The id of the PE that detected the max value

**FIGURE 21.8**

The processing element.

corresponding to the longest match becomes the pointer to the start of the match, which is also transmitted along with the max. The hardware design of the PE, as shown in Fig. 21.8, consists of three parts that perform the above three tasks: the upper module, the central module, and the lower module. For more details, the reader is referred to [64].

21.3 IMAGE COMPRESSION HARDWARE

Digital images require an enormous amount of space for storage. For example, a single-color image with a resolution of 1024×1024 picture elements (pixels) with 24 bits per pixel would require 3.15 MB in uncompressed form. Thus, image compression is critical in large image databases as well as in applications where images are being transferred over the network. Lossless image compression algorithms are especially important in medical and space applications while most other applications can afford a certain amount of loss. The lossless methods for image compression can be classified as statistical coding techniques such as Huffman and Fano and universal codes such as arithmetic coding, Lempel-Ziv codes, and other block level template matching methods. Lossy image compression often presents trade-offs between the amount of compression and the fidelity required in reconstruction and thus is interesting and challenging. Numerous types of methods and their variations can be found in the literature. Lossy methods can be typically classified based on the domains, such as spatial, frequency, and time domains, or the techniques

they are based on, such as transforms, subband coding, wavelets, quantization, and differential encoding.

Image compression methods are computationally intensive and ideal candidates for hardware implementation. A large number of works have appeared in the literature on hardware architectures and implementations for different image compression methods. The JPEG standard is used for still-picture compression and the MPEG standards are used for moving pictures requiring intraframe as well interframe coding. Since the JPEG and the MPEG standards were announced, several different VLSI architectures have been proposed and commercial chips implementing them have been available since the early 1990s. The majority of the hardware implementations for image compression have targeted the following methods: the transform methods such as the DCT, DFT (Discrete Fourier Transform), wavelets, vector quantization, and the JPEG standard itself. The first JPEG standard used DCT and DPCM (differential encoding) for the transform part and a combination of run-length, Huffman coding, and arithmetic coding for the entropy encoding part. While many hardware products implemented the baseline JPEG standard, there is no commercial hardware for the more recent JPEG 2000 standard. For detailed information on the algorithms and architectures for various image compression methods, the reader is referred to a recent book by Bhaskaran and Konstantinides [87]. Recently, the trend in the hardware market has moved more toward video compression products and image compression being packaged as part of the video standard that is implemented. Keeping that in mind, here we provide only brief pointers to a few examples of hardware approaches to image compression and move on to a more detailed treatment of video compression hardware in the next section.

21.3.1 DCT Hardware

It should be noted that two-dimensional DCT computation can be implemented as a sequence of two one-dimensional DCTs, which is commonly referred to as the separability property. This approach is simpler to implement in hardware. It was shown by Haralick [74] that the DCT of N points can be computed using two N -point FFTs by exploiting the symmetry of the inputs. Later, it was shown in [12] that the DCT can be obtained more efficiently by computing just the real part of the first N coefficients of the $2N$ -point DFT. The computation of 8-point DCT needed for JPEG can be replaced by 16-point DFT computation followed by scaling. An optimum form for 16-point DFT was developed by Winograd [81] and was later adapted for 8-point DCT by reducing the computation by using the symmetry property [92]. A modification reducing the number of 2's in complement operations can be found in [57]. Recently, several new and improved VLSI designs for DCT, targeting high performance as well as low power using principles such as multirate arrays and asynchronous pipelines, have appeared in the literature [7, 22]. Due to the extensive use of DCT in various applications that demand real-time processing, numerous VLSI chips have been designed and built by both academia and industry. For a more detailed treatment of DCT and the related commercial VLSI chips, the reader is referred to Ref. [50].

21.3.2 Wavelet Architectures

Several VLSI architectures for one-dimensional and two-dimensional discrete wavelet transforms (DWTs) and continuous wavelet transforms have been proposed in the literature. A detailed survey of these can be found in [17]. In particular, the 2D DWT is used in the JPEG-2000 image compression standard and is computationally intensive; its VLSI realization is still an active area of research. Lewis and Knowles [8] used the 4-tap Daubechies filter to design 2D architectures. Two architectures that combine word-parallel and digit serial computations were proposed in [48].

Several systolic and SIMD arrays for 2D DWTs were proposed in [15–17, 34, 37, 39, 59]. Some recent works can be found in Refs. [65, 78]. Since there are too many contributions in this area, we are providing pointers to only some of the important ones.

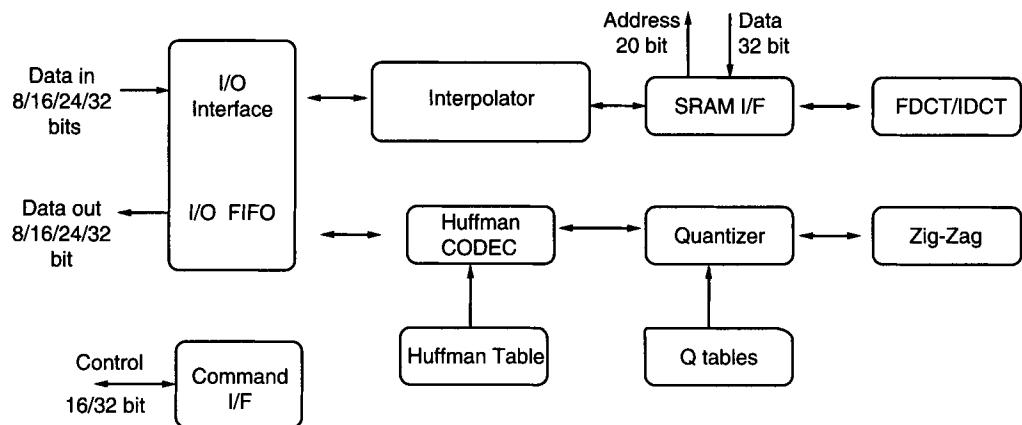
21.3.3 JPEG Hardware

Several special-purpose VLSI chips implementing the JPEG baseline compression standard have been built and successfully commercialized. The majority of earlier image products were from C-Cube, LSI Logic, Zoran, Fujitsu, Intel, and Atmel [87]. Intel's i750 video processor [35, 36] consisted of two chips, the 82 750-PB pixel processor and the 82 750-DB display processor. The pixel processor can be programmed to implement the JPEG compression standard. C-Cube developed two processors, CL560 and CL550 [18]. CL550 operates at 35 MHz; data sustenance was at the rate of 2 MB/s and the compression rate was 30 fields/s. CL560 was installed with enhanced video applications. The sustenance was 60 MB/s. The chip could handle 601 frames in real time. LSI Logic announced a chip-set for JPEG compression that consisted of an L64735 DCT processor, an L64745 JPEG coder, and an L74765 color and raster-block converter and could process still-image data at up to 30 million bytes per second (Mbps). LSI Logic's JPEG chip-set is described in [53, 54]. In July 1993, LSI Logic announced the L64702 single-chip JPEG coprocessor designed for graphics and video applications in personal computers, engineering workstations, and laser printers [1]. The chip was capable of compressing and decompressing data at rates up to 8.25 million bytes per second with an operating frequency of 33 MHz. The architecture of JAGUAR, which implemented the various logic blocks of the entire JPEG compression standard as a linear superpipeline to achieve 100 million bytes per second throughput, was described in [57]. Zoran offered ZR36040 and ZR36050. ZR36040 works with an external DCT processor and controller. Data processing could vary from 15 to 21 MB/s and could process 601 CCIR frames in real time. The ZR36050 JPEG chip was designed as a superset of ZR36040, which had a video encoder and decoder in a single chip. Fujitsu developed MB86357A, which supported color components with either 8 or 12 bits/pixel with a 20-MHz clock. Atmel targeted a low-power, low-cost chip suitable for camera or video applications.

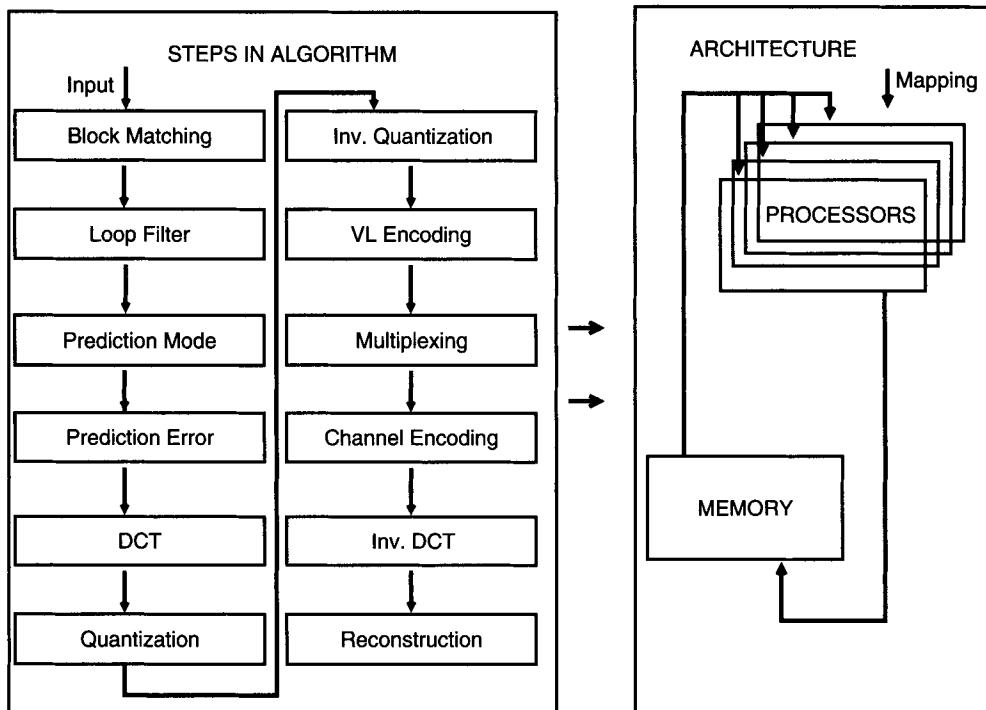
In the present scenario, image compression products are seldom developed. Still-image compression is performed by the video compression chip-sets. We found one JPEG image processor by Oak Technology Inc. The relevant features are discussed below. The product PM-36 is a fixed-function iCODEC for high-speed color and grayscale image data compression. The data sustenance is 110 Mbps. This chip can operate at 80 MHz (for raster/block mode) or 110 MHz (for block mode only). It uses four loadable Q-tables and two loadable Huffman table pairs. The chip has internal PLL and operates at 3.3 V with 5-V tolerant I/O. The JPEG standards are ISO IS10918-1/10918-2 for baseline and color images. The chip has internal buffers of 256 bytes on input and output ports. I/O's can transfer up to 280 Mbps. Figure 21.9 depicts the block diagram for the PM-36 image processor.

21.4 VIDEO COMPRESSION HARDWARE

In recent years, video communication has become an essential part of a wide range of applications. Cheap digital storage media and an abundance of digital transmission channels have triggered a huge growth in video communication. In response to the wide range of applications, various standards for coding and transmission have been proposed by the ISO and ITU groups. MPEG standards such as MPEG-1, MPEG-2, and MPEG-4 are commonly used in video communication, whereas H.261 and H.263 are standards specific to teleconferencing. Video compression

**FIGURE 21.9**

The JPEG image processor by Oak Technology Inc.

**FIGURE 21.10**

Functional specification in a video compression technique [67].

standards are based on a unified underlying coding scheme in which motion-compensated inter-frame prediction is used to reduce redundancy in consecutive frames and is combined with block-based transform coding to allow selective removal of irrelevant information through adaptive quantization and entropy coding [66]. Specifically, the tasks involved are DCT, motion estimation, quantization, and variable-length coding. In MPEG-4, object-based coding is additionally required for encoding video objects. The various tasks are shown in Fig. 21.10. Since these tasks are common to the various standards, the architectural implementation of these tasks is of

much interest. Two basic architectural choices for hardware implementation of video compression are (i) dedicated architectures that try to maximize performance and (ii) programmable architectures that maximize flexibility in terms of adapting to different tasks as required by different standards leading to a more cost-effective system. Dedicated algorithms use the greatest amount of parallelism and exploit completely the regularity and features of the algorithm and hence they can meet the performance criterion with a lower silicon area. DCT and motion estimation as well as entropy coding are often implemented this way. Programmable architectures are preferred when cost and flexibility are of primary concern.

2D-DCT (two-dimensional DCT) can be used as a frequency transform with L^2 computations on an $L \times L$ image block. By decoupling the 2D-DCT into two 1D-DCTs (one-dimensional DCTs) the computations can be reduced to order $2L$. 1D-DCTs are fairly common, and well-known architectures exist in the literature [63, 68, 80, 86]. Fast DCT algorithms have also been proposed by many researchers [3, 13, 70]. Alternatively, the multiplication is completely omitted by using pre-calculated lookup tables stored in ROM [62]. For motion estimation, block-matching algorithms [14, 38, 41, 85] are used to estimate the motion between consecutive frames. The Mean Absolute Distance (MAD) is used to estimate the motion for each $N \times N$ block in the frame. Several alternative approaches are proposed based on signal flow graphs [67]. 2D exhaustive block matching is implemented on a 2D array architecture. Hierarchical block matching has also been proposed [47, 49, 51]. Apart from the function-specific architectures, dedicated implementation of video compression has been widely reported [1, 68, 76, 83]. For example, Ruetz and Tong [68] used a dedicated seven-chip architecture for full-motion video compression. The chip-set has a separate motion estimator, DCT block, quantizer, etc., performing the full compression tasks.

Programmable architectures are more flexible than dedicated architectures and can perform various algorithms on the same hardware. Algorithms are implemented in software. General-purpose hardware does not provide adequate performance for real-time video. Generally, parallelism in the task, instruction, and data levels is exploited to enhance performance [33, 84]. Some strategies typically used for enhancement of performance are (i) increasing clock frequency through pipelining, (ii) parallelizing the datapath to exploit instruction level parallelism as in superscalar and VLIW architectures, (iii) using concurrently working coprocessors, (iv) using array processors, and (v) using SIMD and MIMD parallel models of architecture. Video coprocessors are the current trend in the commercial products for enhancement of performance retaining the programmability [82]. Fig. 21.11 depicts a coprocessor-based architecture implementation that is targeted to provide a cost-effective solution for a low-bit-rate application [89]. The design consists of a control module and a block-level coprocessor along with other I/O and memory modules.

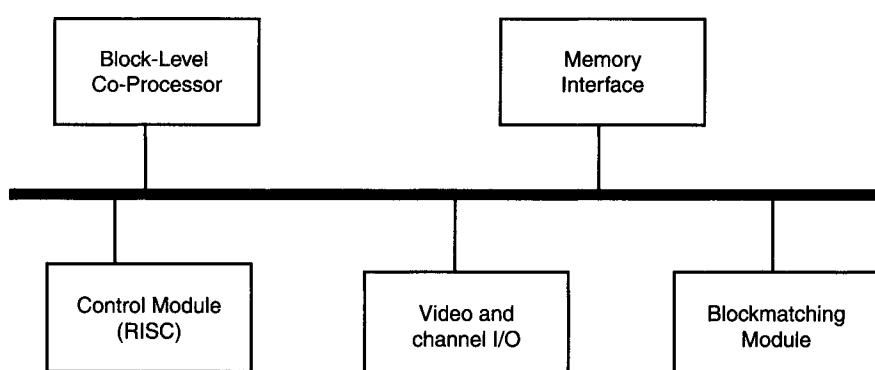


FIGURE 21.11

Co-processor-based architecture proposed by Gehrke *et al.* [89].

This architecture can implement the variable-quantization step. The block-level coprocessors perform DCT, IDCT (inverse DCT), and quantization with two parallel datapaths.

Since numerous architectures exist, we first briefly outline different architectures to point to the current trend in video compression research and then discuss a few examples in more detail.

As an example of programmable architectures, we intend to highlight features from an MPEG-2 4:2:2@HL encoder (developed by Mitsubishi R&D) architecture, which is a hybrid architecture [30]. The single chip performs video and audio encoding and performs system multiplexing of MPEG-2 main profile at main level (MP@ML). Moreover, this chip can be extended easily by parallel processing for MP@HL or 4:2:2@HL video encoding. The system consists of a VLIW-type two-way SIMD processor [30] with communication ports. The performance is 300 MIPS at 162 MHz. This encoder can handle SDTV (standard-definition television) video formats with a single chip. However, for HDTV (high-definition television) format more than six times as many computations are required. Hence the video is segmented and six parallel units work together with the help of two 8-bit communication ports. The motion compensation blocks for MPEG-2 use high computational power and this cost is eliminated by using good motion estimation in the encoder chip.

Berekovic *et al.* [55] proposed the TANGRAM coprocessor aimed at system-on-chip applications for MPEG-4 video. The processor can composite the scenes at the display. This work involves real-time warping and alpha-bending of multiple full-screen video textures. The RISC processor works independently of the decoder. The major computations performed are blending, warping, address computation, and interpolation. VHDL implementation of TANGRAM shows that a 100-MHz clock is achievable with 160K gates, with 350 Kbits of memory, and with consumption of 1 W of power with 0.35- μ m technology.

Badawy and Bayoumi [88] proposed a hybrid object-based video motion estimator by using function-specific architecture. A 2-D mesh created by the architecture captures the dynamics of the video objects, and the architecture relies on a block-matching core to generate motion vectors. The video object is represented by a 2D triangular mesh. The mesh-based motion estimation algorithm works as a 10-state FSM. Using 0.6- μ m CMOS, the power consumption is less than 0.5 W.

Chang *et al.* [28] proposed a bitstream parser suitable for MPEG-4 applications. Based on a bitstream analysis, only seven types of parsing instructions are needed. The core of the architecture consists of three major units (implemented by dedicated hardware implementation): (i) the functional unit, (ii) the memory management unit, and (iii) the instruction decoder unit. The functional unit performs codeword decoding, arithmetic, and logic computation. The memory management unit works to store decoded data and the parsing instructions. The instruction decoder decodes the instructions and generates data for the functional unit and address generator.

Sriram *et al.* [69] proposed an MPEG-2 decoder microprocessor (MAJC) that uses VLIW instructions suitable for multimedia applications. A high-speed VLSI architecture for discrete wavelet transforms was proposed by Chang *et al.* [79] for MPEG-4. Liu [60] proposed an MPEG decoder for embedded systems applications where performance may be relaxed to achieve a simpler and more compact hardware implementation. Features from bus-based and pipeline-based architectures are used to implement simple hardware. Lin *et al.* [19] proposed a low-power design for an MPEG-2 decoder in which Gray code encoding is proposed for increasing the correlation of the bus data transfer to reduce switching activity and hence achieve power reduction.

21.4.1 Some Detailed Examples

Due to the vastness of the literature in this area, we elaborate on only a few recent works as examples, to provide more details that will aid in the general understanding of video architectures.

21.4.1.1 Architecture for Compressed Bitstream Scaling

The idea of scaling the bitstream is generally used to reduce the rate of the constant-bit-rate-encoded bitstream. The applications that use scaling are video-on-demand, trick-play track on digital video tape recorder (VTRs), and extended play recording on digital VTR. The various scaling techniques have different hardware complexities and performance trade-offs. We concentrate on the various bitstream scaling architectures discussed by Sun *et al.* [31]. They perform scaling irrespective of the way in which the stream was encoded. The various available architectures are (i) scaling higher frequencies, (ii) scaling by requantization, (iii) scaling by reencoding the reconstructed pictures with motion vectors and coding decision modes extracted from the original high-quality bitstream, and (iv) scaling as in architecture (iii) above, but with new coding decisions based on reconstructed pictures. The first two architectures, (i) and (ii), are important for VTR applications and the last two architectures, (iii) and (iv), are important for VOD applications.

In architecture (i), bitstream scaling is performed by scaling higher frequencies. This is achieved by using two parallel paths in the architecture. In one path, a Variable-Length Decoder (VLD) parser is used to obtain the codeword length and a bit-allocation analyzer collects all the AC bit-counts for every macroblock in that frame. The scaling factor is determined based on profiling these AC bits, and the linearly scaled profile is sent to the final rate-controller block, as shown in Fig. 21.12. The other path consists of a delay proportional to the first path and another VLD parser that delivers all the codeword blocks to the rate controller. The rate controller accepts only when the running sum of DCT codeword bits exceed the scaled profile value less the EOB bits. The rest of the DCT codewords are excised.

Architecture (ii) has variable-length coding and additional quantizer and dequantizer blocks. In this architecture, shown in Fig. 21.12, the rate-controller functions are based on a requantization scheme. The DCT coefficients are first dequantified to a finer grain quantization and then requantified back with a coarser quantizer scale using the profiling information from the first path, adaptively determining the quantization steps.

In architecture (iii), the motion vector and macroblock coding decision blocks are first extracted, and reconstructed pictures are obtained by normal decoding procedures. Scaling is then performed by reencoding and reconstructing, using the motion vector and macroblock decision extracted earlier. Applying reencoding after a full decoding eliminates the need for separate motion estimation and decision computations. Figure 21.13 describes architecture (iii).

Architecture (iv) is a modification of architecture (iii), where new macroblock decision modes are computed during the reencoding of the reconstructed picture. It is theoretically important to obtain decision modes from the scaled bitstream rather than from the initial bitstreams.

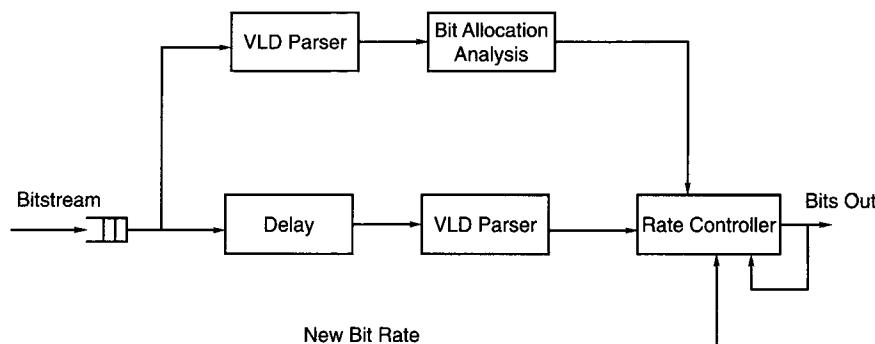
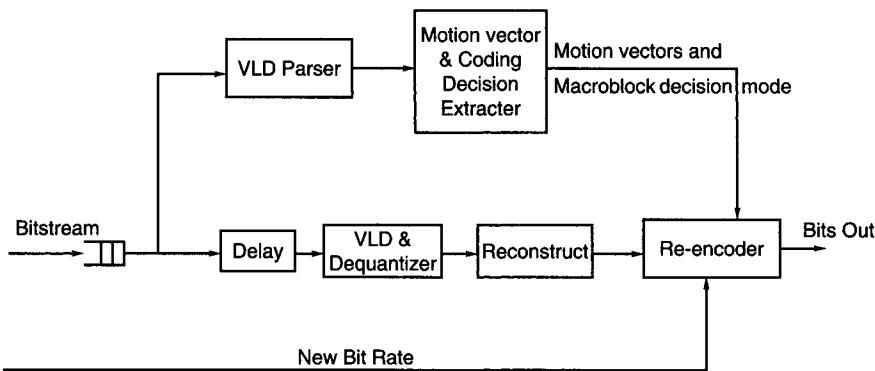


FIGURE 21.12

Bitstream scaling by cutting high frequency as proposed by Sun *et al.* [31].

**FIGURE 21.13**

Bitstream scaling by reencoding using original motion vector and decisions as proposed by Sun *et al.* [31].

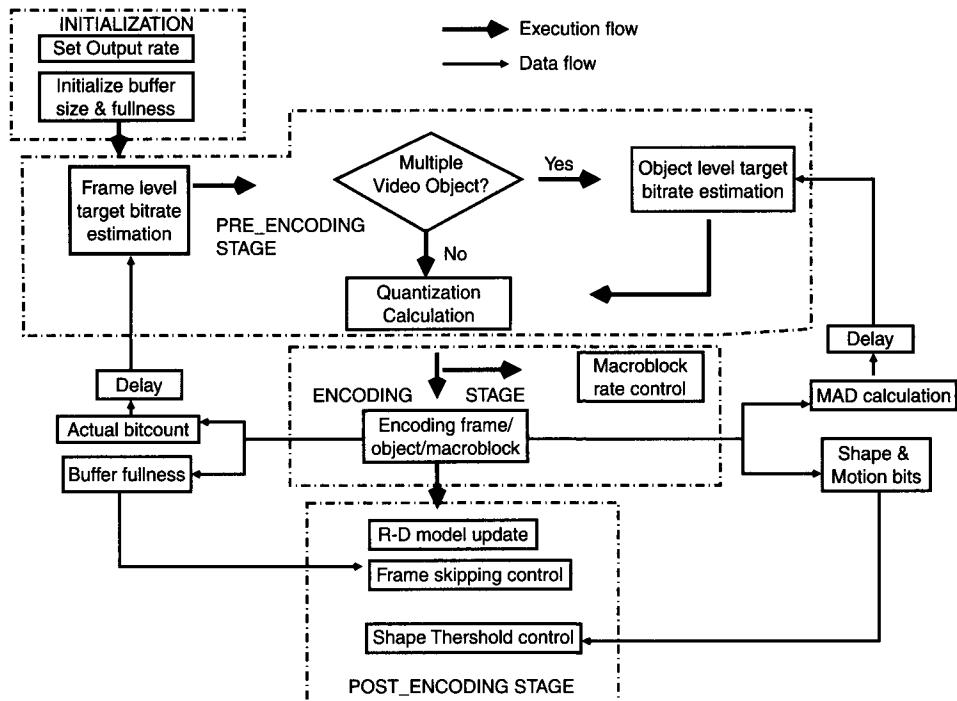
21.4.1.2 Transportation of MPEG-4 on the Internet

Let us now focus on a recent contribution by Wu *et al.* [24] on transporting MPEG-4 video over the Internet. Any transporting mechanism must include source-rate adaption, packetization, a feedback system between the sending and the receiving ends, and an error control scheme. This work focuses on (i) an end-to-end feedback controller that helps estimate the network bandwidth by the packet-loss information, (ii) an MPEG-4 video encoding rate controller that adaptively changes the output rate according to the network bandwidth, and (iii) a packetization scheme utilizing MPEG-4 video-object plane features. We highlight some important ideas regarding the above three aspects here.

The feedback controller is designed according to the RTP/RTCP standard. The network bandwidth is estimated indirectly through a feedback from the receiver end. The feedback algorithm modifies the bandwidth according to the packet-loss ratio sent back from the receiver. The output rate of the MPEG-4 is adjusted through the feedback scheme to keep the the packet-loss ratio below a required threshold value.

The role of the Adaptive Encoding Rate Controller (ARC) is to maintain the output rate of the encoder based on the feedback controller estimates. Figure 21.14 depicts the structure of this ARC unit. The ARC uses an accurate second-order R-D model that introduces two important components, namely, M, which is MAD, and H, which denotes bits for header, motion vector, and shape information in the quadratic model. By introducing M (MAD), this method ensures scalability with coding complexity. The first stage, initialization, as shown in Fig. 21.14, sets the initial buffer size and output rate. In the second, preencoding stage, the target bit-count is first estimated at three different levels, namely, the frame, object, and macroblock levels, to improve estimation accuracy. The bit-count is further adjusted based on the buffer information for each video object, and then quantization parameters are computed. The third stage, encoding (Fig. 21.14), uses the measurement of actual bit-rates. Macroblock-layer rate control may also be activated at this stage. The final, postencoding stage is responsible for updating the R-D model parameters. At this stage, a decision is made to balance the usage between shape information and texture information. Moreover, buffer overflow is controlled by skipping some frames.

Wu *et al.* also proposed a packetization scheme appropriate for MPEG-4 transportation [24]. The proposed packetization algorithm uses a packet size that is the minimum of the maximum transit unit and the current VOP (Video Object Plane) size. When the VOP is too large to fit into one packet, then it is broken up into multiple packets. The proposed algorithm is designed to minimize the number of packets and also the dependency among packets. If a single VOP does not fit into a packet, a maximum number of Motion Blocks (MBs) are collected in one packet.

**FIGURE 21.14**

Adaptive rate controller for transportation of MPEG-4 video over the Internet as proposed by Wu *et al.* [24].

The VOP header information is copied into all the packets that are part of the same VOP. MBs from two VOPs are never put into a single packet, even if space is available, to reduce conflicting header information.

21.4.1.3 MPEG-4 Video Codec by Toshiba

Recently many works have focused on MPEG-4 video and audiovisual codecs [40, 91]. We will focus on hardware implementation of a complete MPEG-4 video codec by Toshiba and discuss various components in the codec.

The MPEG-4 architecture model developed by Toshiba [58] consists primarily of a 16-bit multimedia-extended RISC processor, MPEG-4 audiovisual LSI containing three 16-bit RISC processors and 16-bit DRAM, and a dedicated hardware accelerator suitable for IMT-2000 applications. The objective of the architecture is to achieve higher programmability with low power consumption. The RISC processors can perform complicated tasks on the hardware accelerator such as bitstream syntax processing, rate control, and parallel operation control [58]. Moreover, the hardware accelerator itself performs critical tasks such as motion estimation, motion compensation, and DCT. The accelerator has its own memory and can perform independently of the RISC processor, making parallelization possible. Low power consumption is feasible due to the parallelization [58].

In Fig. 21.15, we explain the functionality of the video codec architecture. The RISC processor has a 4-KB direct-mapped instruction cache (I-cache) and 8 KB of local memory. Variable-length coding and decoding are performed for the DCT coefficients by the processor. The processor also analyzes and synthesizes the bitstream, monitors an intra-AC/DC prediction, and monitors the control of the hardware accelerators. The instruction cache is responsible for the reduced area of

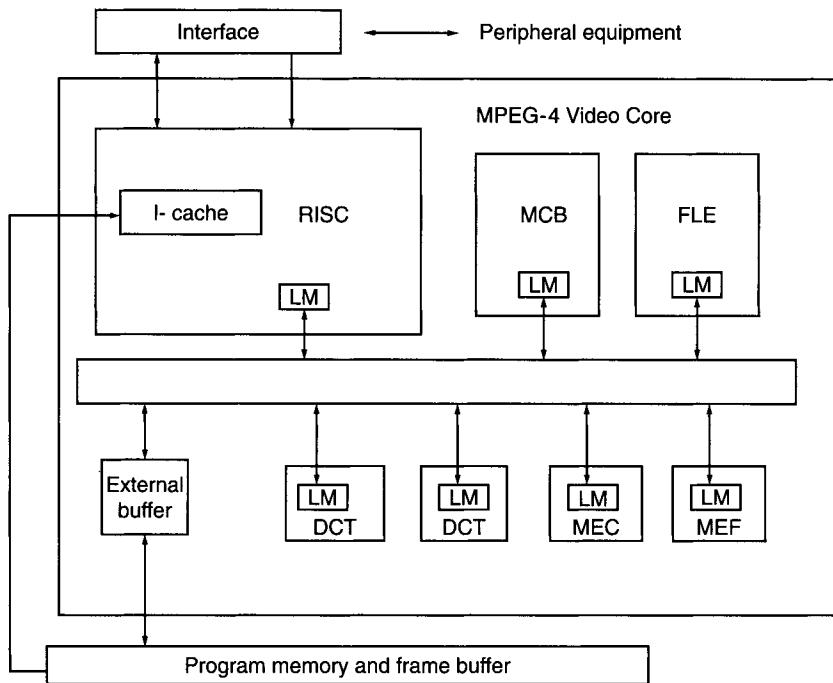


FIGURE 21.15

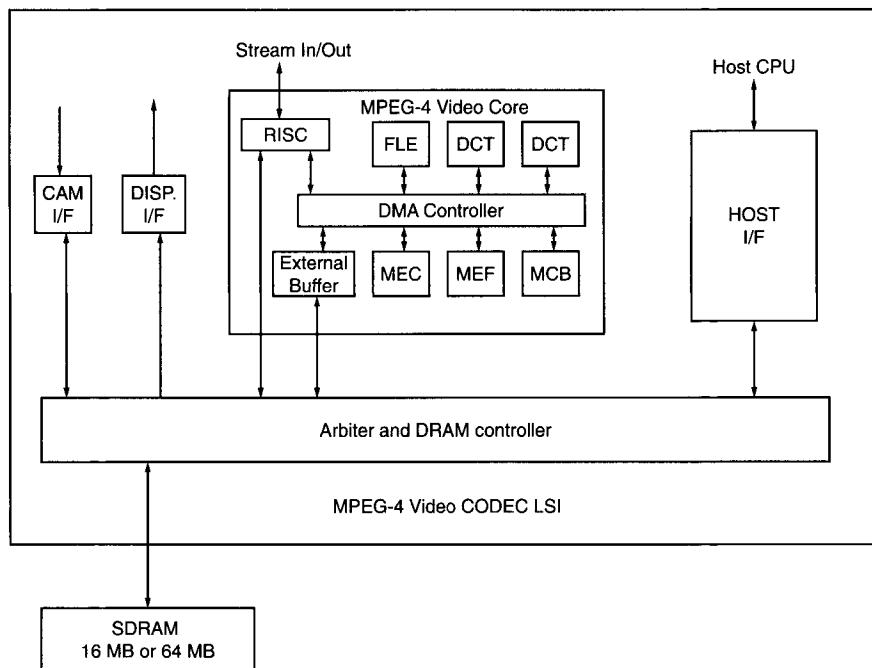
MPEG-4 architecture for video core LSI [58].

the chip (since on-chip SRAM is not required) and reduced memory access. The DCT block either performs DCT and quantization (Q) or performs IDCT with IQ (Inverse Q). Both the DCT blocks can operate simultaneously for speed-up. The motion compensation block MCB, MEF, and MEC handles the motion estimation with a variety of search windows using a three-step hierarchical search.

The FLE block is used for postfiltering the hardware accelerator for better image quality. The frame buffer and firmware are stored in the external DRAM. The hardware accelerator uses local memory for parallel operations. As a consequence of the reduced external memory access, effective memory access time is also reduced. The total local memory is 5.3 KB, which occupies a reasonably small area. DMA controls all the memory accesses, and power-efficient multiplexors are used for data transfer, rather than buses with large capacitance.

The processing of the video core is performed by an external firmware on the central processing unit (CPU). A few threads run concurrently to generate the desired amount of parallelism. The encoding thread handles six blocks of data with 6-stage pipelining for Y0, Y1, Y2, Y3, Cb, and Cr. By this pipelining, 15 frames of encoding and decoding can be processed with QCIF (quarter common intermediate format) at a 60 MHz/s clock rate.

Effort has been made in this architecture to reduce power via various means. The architecture also uses clock-gating to reduce clock power. When the hardware accelerators are idle, the clock circuit is stopped. The RISC processor has its own instructions to go into sleep mode by stopping its own clock. Embedded DRAM cuts down the power consumption in I/O circuits. Moreover, techniques to optimize the page and word size are used to obtain better power efficiency. A low-power motion estimator has been adopted using techniques similar to those outlined in Ref. [2]. The motion estimator achieved almost 50% of the total power savings.

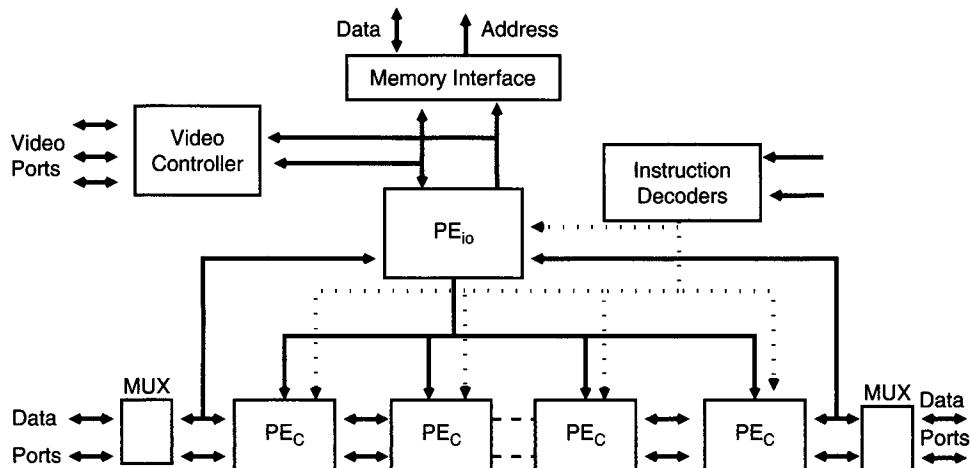
**FIGURE 21.16**

MPEG-4 architecture for video codec LSI [58].

The MPEG-4 video codec, as depicted in Fig. 21.16, consists of the RISC core described earlier and some peripherals. A CIF CMOS image sensor and an LCD controller are directly connected to the inputs and the outputs. The codec uses an ordinary bus interface along with a bitstream serial interface for multimedia applications. Additional memory (16- or 64-Mbit SDRAM) can be connected for program storage and frame storage.

21.4.1.4 The MOVIE Project

Another interesting architecture involves hardware support for software-only real-time video processing as proposed by Charot *et al.* [26]. This approach, termed MOVIE, utilizes a high level of parallelism. The computation and data access in algorithms such as DCT, quantization, and motion estimation are regular [26] and hence SIMD with a linear array processor is used. The MOVIE chip consists of the following components, as shown in Fig. 21.17: (i) four 16-bit data ports, (ii) three 16-bit video ports, (iii) two instruction ports to send to the PE_c processor array for computation and to the PE_{io} I/O processor for I/O operations, and (iv) a memory interface for external memory. Each PE_c is a 16-bit processor and the PE_{io} is a 32-bit I/O processor with the same internal architecture for low-overhead software development. The instruction is controlled by 40-bit instructions, and decoder units are used internally for decoding. Since SIMD allows all processors to use a single instruction, one I/O processor and one controller are sufficient. Instructions can also be pipelined to generate greater throughput. One communication operation between the neighboring processors in the systolic array of processors can be done simultaneously with the computation. The instruction formats use two independent fields: one for computation and one for communication. The MOVIE approach relies on C-Stolic software support for real-time video processing.

**FIGURE 21.17**

Architecture for real-time video processing [26].

21.4.2 Commercial Video and Audio Products

We focus on the industry-released compression products available for video and audio compression, most of which come as single-chip solutions. We combine the descriptions of audio and video products since they are supported together in most multimedia systems. Products differ in terms of the standards they support, input and output formats, and compliance with various application standards. Products support different standards, such as MPEG-1, MP@ML MPEG-2, and 5.1 Dolby Digital. The inputs can be bitstreams (packetized elementary or elementary streams) in serial or parallel formats. The output format can be of various types, such as NTSC or PAL. The products comply with DVD, Digital Video Broadcasting (DVB), Digital TV, HDTV, or SDTV applications. Recent trends also indicate that most codecs are for combined audio and visual applications, with synchronized audio and video units. An important reference in this area is the book by Bhaskaran and Konstantinides [87]. Here, we briefly outline various products, examining a few in further detail.

Earlier audio products were either dedicated or programmable DSPs with glueless interface to memory and a host controller. Crystal semiconductors produced the CS4920 programmable DSP chip for MPEG-1 or AC-2 audio decoding. CS4922 from the same manufacturer could support both MPEG-1 and MPEG-2 layers 1 and 2. L64111, developed by LSI Logic, is a two-channel MPEG audio decoder supporting MPEG-1 and MPEG-2. SGS-Thomson developed the STi4510 MPEG-1 audio decoder, which supports all MPEG-1 sampling rates. Most processors from Texas Instruments (TI) can be run as audio decoders. Moreover, TI produced a dedicated audio decoder, TMS320AV110. Zoran produced programmable audio, DSP, which could potentially handle six-channel AC-3 decoding and two-channel MPEG audio decoding for future generation audio products.

Some of the video products mentioned here are the predecessors of the current products described below. These chips have extensive applications in set-top boxes for interactive TV, HDTV, etc. C-Cube developed the CL480VCD and CL480PC, AVIA-550, and AVIA-502 audio/video decoders. IBM produced chip-sets MPEGSE10, MPEGSE20, and MPEGSE30 for MPEG-2 video encoding. IBM also had an MPEGCD20 MPEG-2 audio/video decoder. TI developed AV7000, which is a complete set-top decoder. It has a 32-bit RISC controller, A/V decoder, graphics accelerator, and traffic interface manager. Some of the other important products were L64002,

L64005, and L64020 A/V decoders from LSI Logic; STi3400, STi3430, and STi3520 MPEG-2 A/V decoders from SGS-Thomson; HDM8211M from Hyundai Electronics; and TC81201F and TC81211F from Toshiba.

We describe a number of current audio and video products in the rest of this section. Most of the information was obtained from the datasheets provided by the vendors. We found that most recent video products come with the audio component as well.

21.4.2.1 C-Cube Video Products

The C-Cube MPEG-2 Video Codec (DVXCEL) is a single-chip MPEG-2 video and system encoder and decoder. This chip can simultaneously encode and decode MPEG-2 video with audio synchronization to make it usable for Digital Video Recorders (DVRs). The device has a 16/32-bit host interface with a DMA target of primary, secondary, and tertiary hardware I/O ports (8 bits each) for bitstream data transfer. The device has 8- or 10-bit 656-video input as well as an 8-bit video output port. The 110-MHz micro-SPARC RISC core consists of hardware for video compression preprocessing, motion estimation and compensation, DCTs, IDCTs, variable-length encoding and decoding, and high-quality video scaling and compositing. This device, developed by C-Cube, uses 8 MB of external SDRAM and has an on-chip controller with a 64-bit-wide interface. The chip consumes 1.8 W of power at a voltage of 2 V. The block diagram is shown in Fig. 21.18.

The ZiVA-5 DVD processor was also developed by C-Cube. ZiVA-5 is a single-chip compatible with DVD-video, DVD-audio, DVD-VR, Chaoji-VCD (CVD), SuperVCD, VCD, CD-DA, and

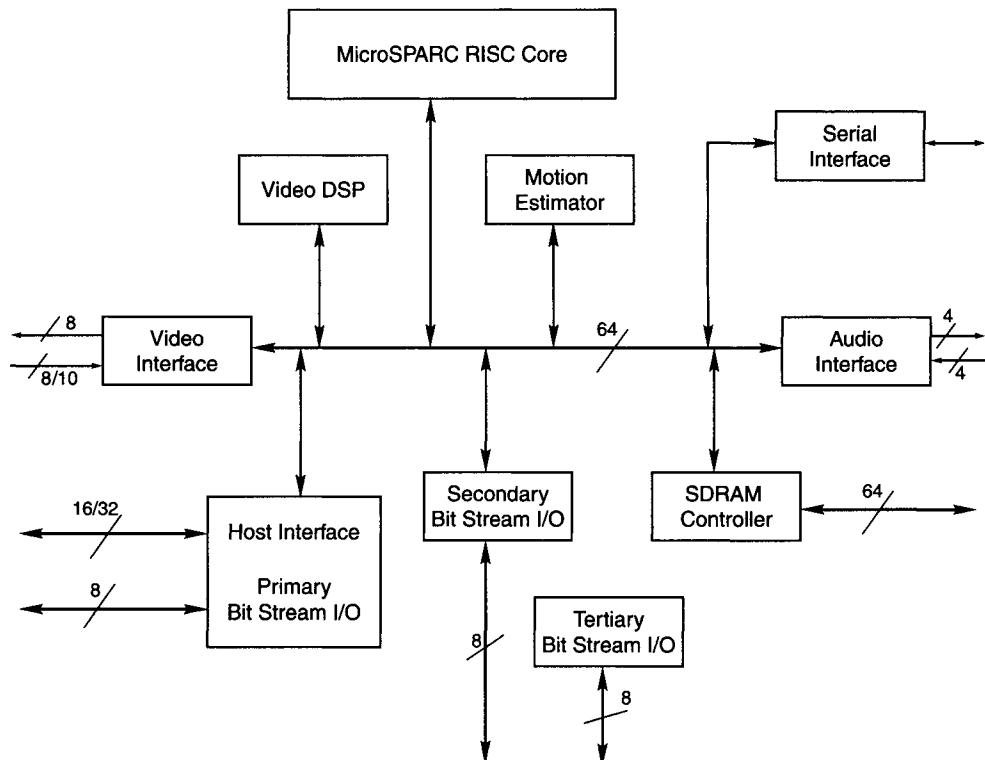
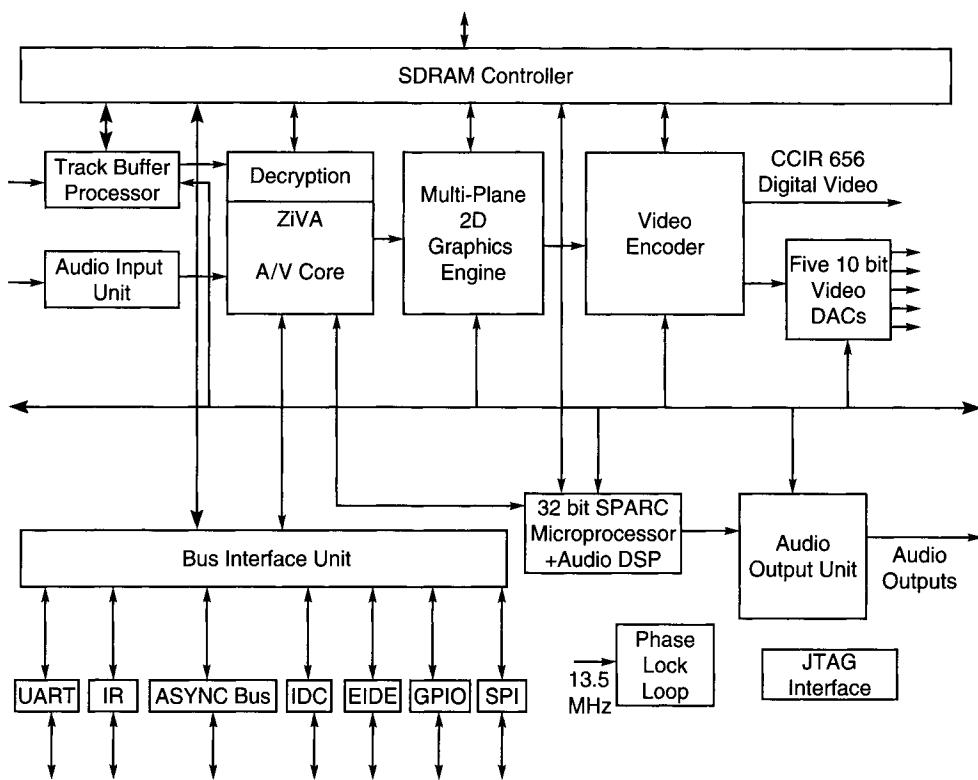


FIGURE 21.18

The DVxcel MPEG-2 video codec by C-Cube.

**FIGURE 21.19**

The ZiVA-5 DVD system processor by C-Cube.

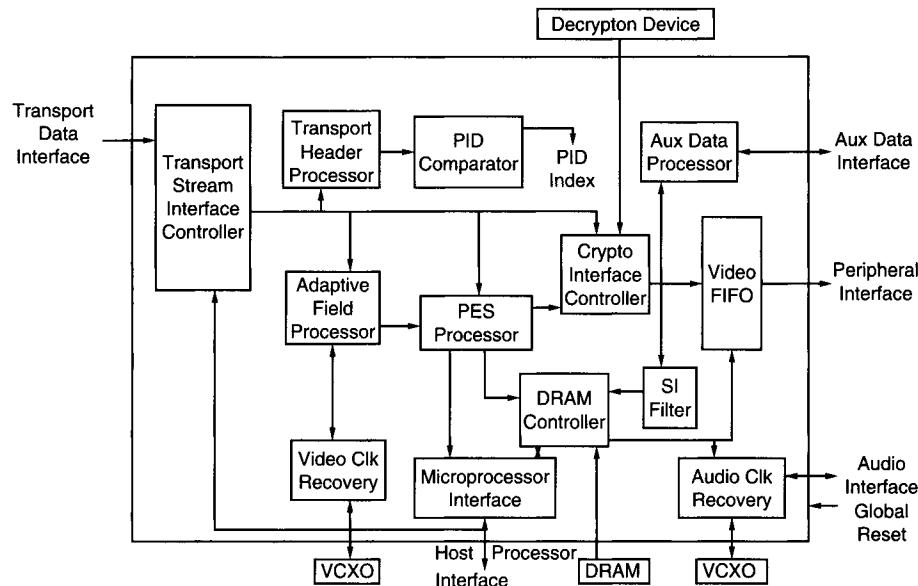
CD-ROM formats. The chip has a dual-audio DSP architecture with a ZiVA audio/video decoder. The chip also includes a high-performance 2D graphics engine and a video encoder with five 10-bit, 54-MHz DACs for NTSC, PAL, and 480P formats. The DVD-audio support includes MLP decode, content protection for prerecorded media, and audio watermark detection. Moreover, ZiVA-5 includes an MP3 audio codec. The block diagram is shown in Fig. 21.19.

21.4.2.2 Lucent System-Layer Decoder

Lucent Technologies developed an AV6220A MPEG-2 system-layer decoder in 1996 that works with multiple audio/video decoders. The device is compatible with system-level DVD requirements and MPEG-2 specifications. This device can handle serial or byte-wise parallel input streams (transport stream format) and may also receive data in transport stream format for CD-ROM applications. The product also provides a clock recovery at 27 MHz and helps synchronize video and audio data. The device uses 0.55- μ m CMOS technology, requires 5.0 V, and consumes less than 1 W of power. The block diagram is shown in Fig. 21.20.

21.4.2.3 LSI Logic Products

LSI Logic developed the L64105 MPEG-2 audio/video decoder in 1997. This decoder chip works for MPEG-2 decoding standards. This chip can be used inside a decoder system chip, which includes other units such as L64108 transport Demux, audio DAC, PAL, and the NTSC format converter. This chip can also be used in set-top box applications or in PC-based MPEG-2 applications. This chip is an improved version of L64005 with enhanced video quality and with

**FIGURE 21.20**

The AV6220A MPEG-2 system-level Demux by Lucent Technologies.

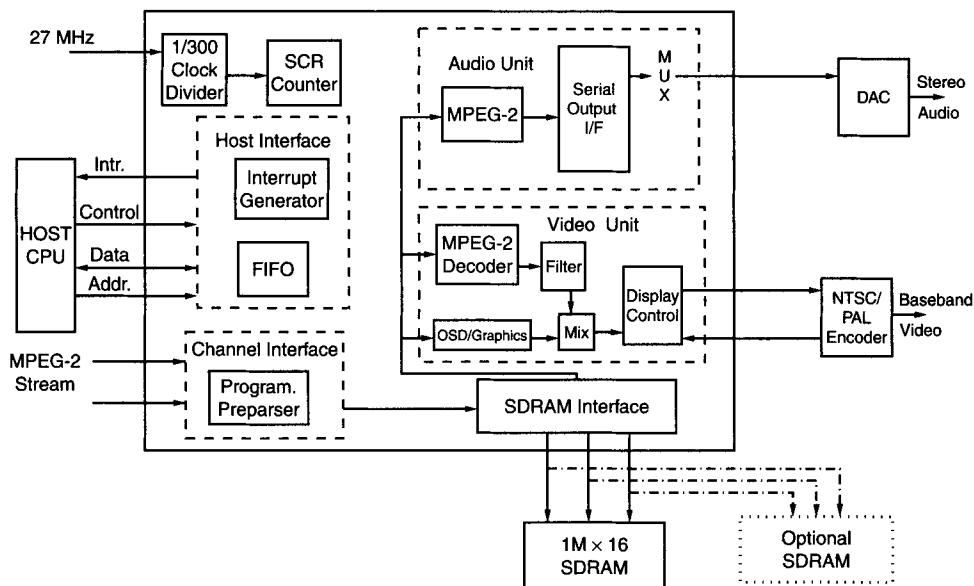
low host-processor overhead. The chip is built using 0.35- μm CMOS technology with a 3.3-V supply voltage.

The video decoding unit is fully compliant with the system-level MPEG-2 standard ISO 13818-2. The decoding unit processes the MPEG-2 bitstream including the MPEG-2 program streams PS and PES. MPEG-1 bitstreams are also decoded including an MPEG-1 system layer. The decoder can handle image sizes up to 720 \times 480 pixels at 30 frames/s for NTSC and 720 \times 576 at 25 frames/s for PAL.

The audio decoder follows MPEG audio decoding standards with support for linear PCM data. It also decodes dual-channel MPEG audio layer I and layer II ISO 11172-3. The bit-rates can range from 8 to 448 Kbps. The sampling frequencies can be 16, 22.05, 24, 32, 44.1, or 48 KHz.

The L64105 chip supports on-screen display (OSD) and display management. The graphics support unit can handle a programmable channel buffer and display buffer size. The chip also has a data- and error-handling unit. The data can be maintained at a rate of 20 Mbits/s for the PS mode and 40 Mbits/s for the PES mode. Input data are in 8-bit parallel mode and output is 8 bits (Y, Cb, Cr) in slave mode. The error-handling unit maintains the video during channel error, whereas the audio unit is muted. The chip also has a cost-effective system implementation with a preparser accepting PES, ES, and PS. The system is directly connected to one or two 16-bit SDRAMs. The chip operates at a 27-MHz clock and can directly interface with available NTSC/PAL encoders and audio DACs. The circuit diagram is shown in Fig. 21.21.

LSI Logic also developed an L64118 MPEG-2 transport controller with an embedded MIPS CPU in 1998. This device has combined set-top box control along with the logic for a communication system and digital broadcast system. The TinyRISC 32-bit CPU provides the transport and system support as well as general-purpose system control. This system integrates with other satellite and cable single-chip channel decoders as well as the L64105 MPEG-2 A/V (audio/video) decoder, which was discussed earlier. THE MPEG-2 transport and system Demux can handle 32 packets simultaneously, including video, audio, and general-purpose data services. The system also supports a DVB-compliant descrambler block with increased security. The external system

**FIGURE 21.21**

The L64105 MPEG-2 audio video decoder by LSI Logic.

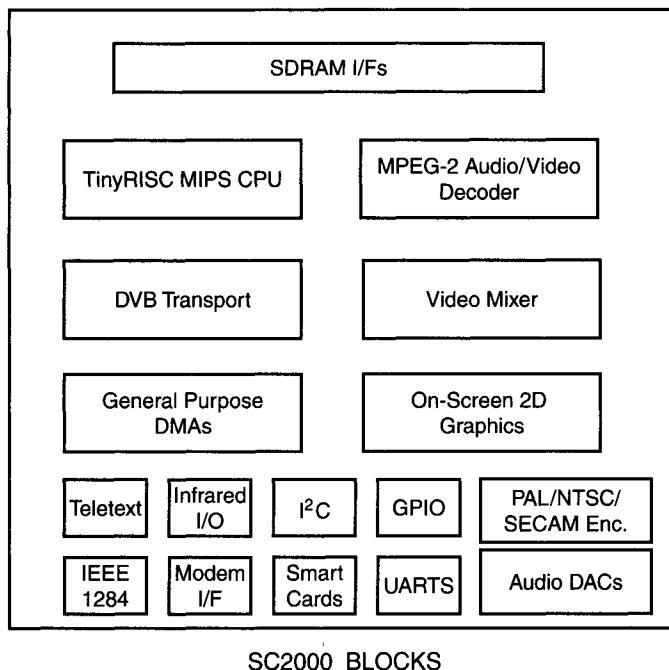
bus communicates through serial, parallel, SmartCard, and interface ports. The memory support can be up to 16 MB using 16- or 64-Mbit SDRAMs. This system is built via 3.3-V G10-p-cell-based technology developed by LSI Logic.

The LSI Logic chip SC2000 provides a multimedia chip (released in 1999) that contains DVB transport and MPEG-2 decoding, a 2D 5-plane graphics engine, a multistandard video encoder, audio DACs, and a RISC MIPS CPU. The chip operates at 108 MHz and includes a unified OSG, CPU, transport memory architecture, and a dedicated A/V memory. This chip complies with ISO/IEC 13818 MPEG-2 MP@ML standards and provides DVD support. The SC2000 chip has a SmartCard interface, a UARTs/modem codec interface, and an I²C compatible interface. This chip provides Dolby AC3 support and external CPU support. It also has general-purpose I/Os, IEEE1248 ports, and a general-purpose DMA controller.

SC2000 uses a dual-memory architecture that helps the performance. The dedicated A/V memory bus provides the high bandwidth without hampering CPU and graphics performance. CPU and OSG memories are unified, delivering higher graphic performance. The block diagram is shown in Fig. 21.22. The SC2005 single-chip source decoder designed by LSI Logic in 2000 has almost the same functional features as SC2000. However, the latter is integrated with NDS ICAM technology accommodating NDS Videoguard conditional access. The SC2005 chip also integrates an IDE/ATA interface.

21.4.2.4 NEC Audio Video Codec

NEC Electron Devices launched two audio-video codecs, μ PD61051 and μ PD61052. Video encoding can be performed at a constant bit-rate or a variable bit-rate. The products support MPEG-2 video, MPEG-1 video, MP@ML, and SP@ML (simple profile at main level). Video decoding is performed using a constant bit-rate or a variable bit-rate. The formats supported are MPEG-2 and MP@ML. The audio formats are MPEG audio layer 2 and Dolby Digital. The products use a program stream/transport stream, DVD, D-VHS (for MPEG-2), and Video CD

**FIGURE 21.22**

SC2000: a single-chip source decoder by LSI Logic.

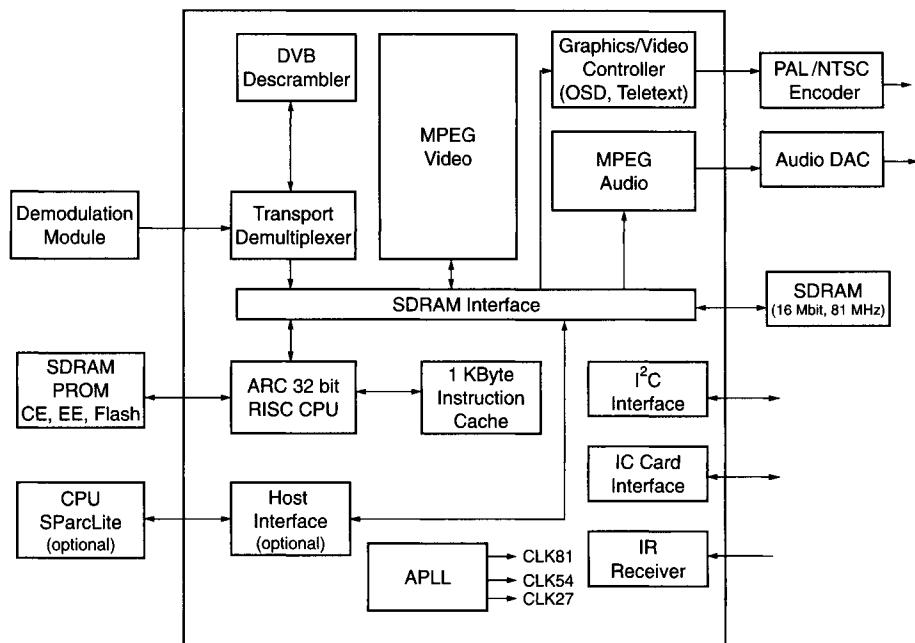
for MPEG-1. μ PD61051 and μ PD61052 have a horizontal resolution conversion (720 pixels to 544, 480, or 352 pixels). The products operate at 27 MHz.

21.4.2.5 Audio Video Products by Fujitsu

Fujitsu developed the MB87L2250 MPEG-2 transport video and audio decoder with 32-bit RISC CPU. This product was released in 1998. This single chip is an audio/video decoder coupled with the transport Demux and a 32-bit RISC CPU. The MPEG decoding and transport Demux is implemented in the hardware. The chip operates at 54 MHz and uses 2.5-frame architecture. The memory requirement is only one 16-bit SDRAM. Along with the RISC CPU, the chip also has a 1-KB I-cache. MB87L2250 also features a DVD descrambler, IR receiver, FR30/68 K/SPARClite processor interfaces, 2-, 4-, 6-, or 8-bit OSD, and a 16-bit programmable general-purpose I/O.

The transport Demux allows parallel/serial input format for the transport stream. The maximum input data rate is 60 Mbps. The MPEG-2 transport stream is ISO/IEC compliant. The MPEG-2 video decoder uses MP@ML, ISO/IEC 13818-2, MP@LL, and SP@ML formats. The MPEG-1 stream is decoded as DVB compliant. The audio decoder features ISO/IEC 11172-3 (MPEG-1 audio) and ISO/IEC 13818-3 (MPEG-2 audio in two channels). The audio decoder works with various sampling rates such as 32, 44.1, 48, 16, 22.05, and 24 KHz. The audio decoder also supports all bit-rates. Both the video and audio decoders have error detection and concealment schemes. Both video and audio decoders support different DACs. The block diagram is shown in Fig. 21.23.

Fujitsu also developed the MPEG2 single-chip audio/video encoder in 2001. This chip integrates a video encoder, an audio encoder, and system encoders. The video encoder works for MPEG-1- and MP@ML-compliant MPEG-2 standards. The video format can be NTSC or PAL. The maximum bit-rate is 15 Mbps. The audio encoder unit follows the MPEG-1 layer 1/2 standard. The maximum bit-rate is 448 Kbps. The system mux operates at a constant or variable

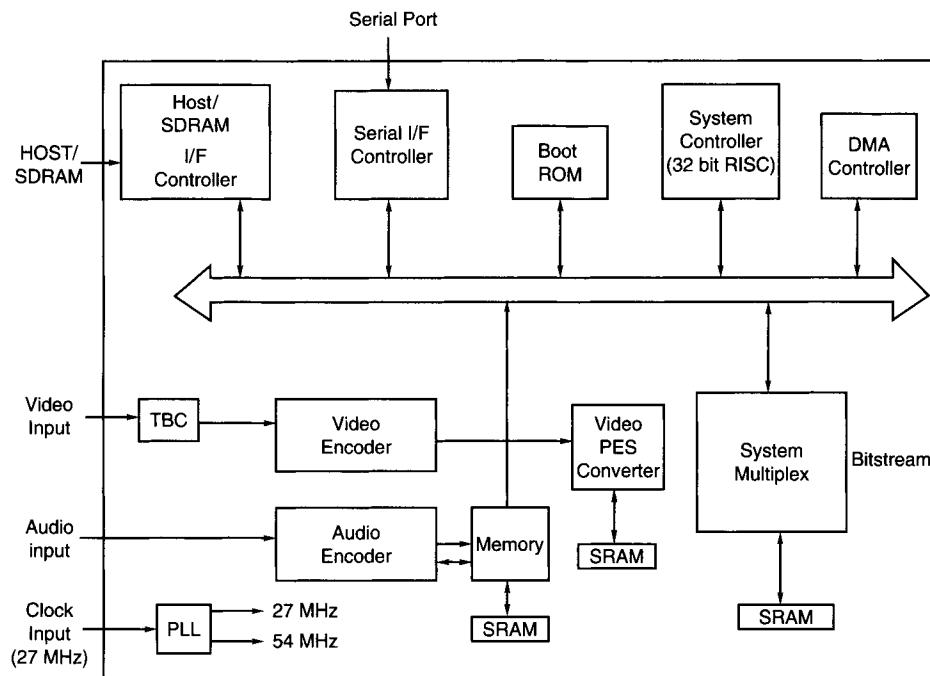
**FIGURE 21.23**

ML87L2250: MPEG2 decoder with integrated 32-bit RISC CPU by Fujitsu.

bit-rate with a maximum of 20 Mbps. The output format can be PS, TS, PES, or ES. This unit has an external memory SDRAM (four 16-Mbit memory chips or two 64-Mbit memory chips). The unit has an internal 32-bit RISC processor with a time base corrector. This chip can be useful for video transmission systems, telephony systems, digital video recorders, and other applications. The block diagram is shown in Fig. 21.24.

21.4.2.6 ATTEL Digital Audio Decoder

ATTEL has developed the AT76C2XX Dolby Digital AC-3 decoder. The decoding is performed on a 5.1-channel Dolby Digital (AC-3) audio stream. The chip supports all Dolby Digital configurations. The chip supports 8 audio channels and also decodes MPEG-2 and AAC. The audio input and output provide 24-bit precision at sample rates of 32, 44.1, 48, and 96 KHz. The chip also receives data from an IEC 958-compatible interface (AES/EBU or S/PDIF) or directly from an A/D converter. All 8 audio channels that the chip supports are compliant with the 5.1-channel Dolby Digital standard. The chip is based on the ATTEL 24-bit DSP. This solution does not require any external SRAM. The audio data output is compatible with the available audio DACs. The chip has 64 pins and requires 3.3 V. The AT76C2XX chip can handle a Dolby Digital bitstream encoded at a rate of 640 Kbps at 48 KHz. Moreover, it can generate audio samples at 96 KHz. The architectural features are a 45-MHz instruction clock with two 24-bit datapaths and one 40-bit datapath with a 16-bit addressing mode. A three-stage pipeline is used. A high degree of parallelism is due to the 40-bit instruction wording. The DSP has 4K × 40 program memory, 8K × 24 X-data memory, and 4K × 24 Y-data memory. The DSP also has two 56-bit accumulator registers and performs multiply, accumulate, and convergent rounding in one clock cycle. DSP has two maskable interrupts one unmaskable interrupt, and two PLLs: one for internal logic clocks and one for D/A converter clocks. This implementation is awaiting the evaluation process by Dolby Digital Laboratories.

**FIGURE 21.24**

MPEG-2 system encoder LSI by Fujitsu.

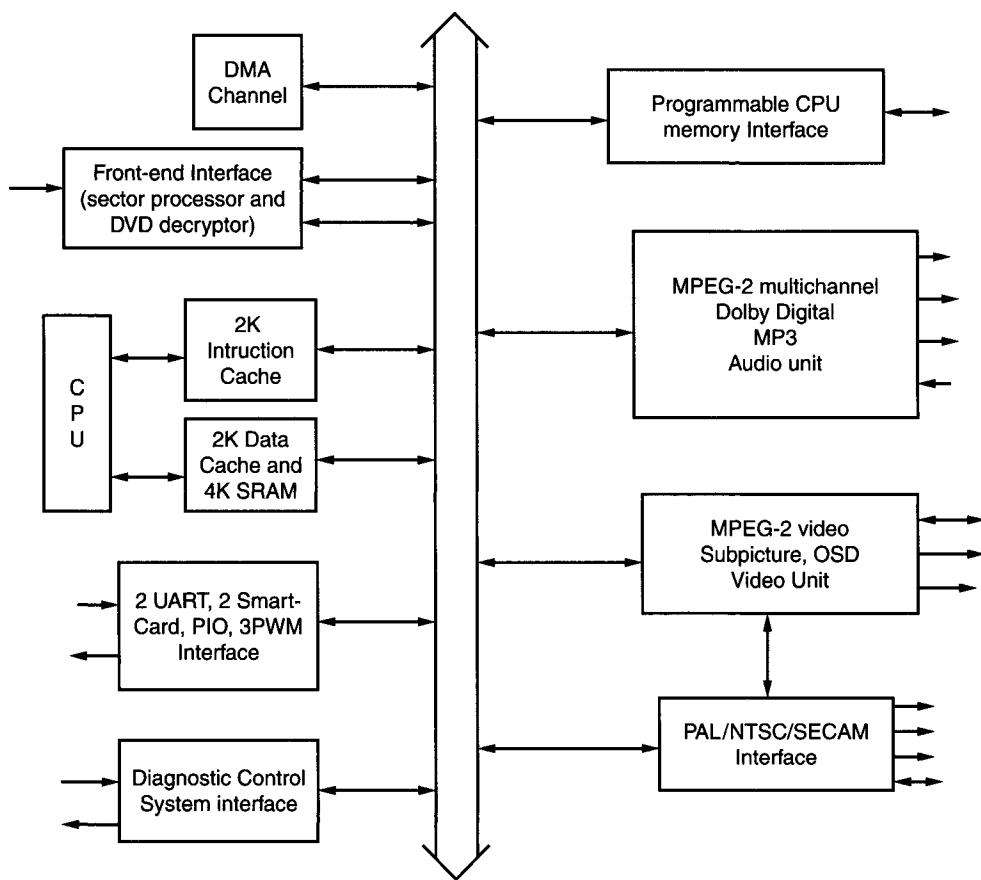
Other ATMEL products include an accelerator for video conferencing and demodulators for digital reception and transmission.

21.4.2.7 Products by STMicroelectronics

STMicroelectronics developed the STi5518 single-chip set-top box decoder in 2001. This chip is designed to be used in pay-TV set-top boxes. It has a 32-bit CPU, a dedicated block for transport Demux and descrambling, video and audio decoders, and low-cost satellite receivers. This chip uses a sector processor and a CSS decryption block in the transport interface and a subpicture decoder in the video decoder. Moreover, the audio decoder handles Dolby Digital audio. These features allow this chip to be used as a DVD-compatible set-top box. The ATAPI interface promotes connection of DVD-ROM and hard disk drives. Thus, live-TV pausing and timeshifting (trick modes) can be handled. This product is compatible with other STMicroelectronics products such as STi5500.

The video decoder supports MPEG-2 MP@ML with zoom-in/zoom-out facilities, PAL to NTSC and vice versa conversion, and DVD and SVCD subpicture decoders. The video decoder also supports 2 to 8 bits/pixel OSD options. The PAL/NTSC/SECAM encoder is interfaced. The output formats supported are RGB, CVBS, Y/C, and YUV with 10-bits. The RISC processor operates at 60 MHz and has a 2-KB instruction cache, 2-KB data cache, and 4 KB SDRAM as an additional data cache.

The audio decoder uses 5.1-channel Dolby Digital MPEG-2 multichannel decoding and 3×2 -channel PCM outputs. The decoder produces IEC60958–IEC61937 digital outputs. The DACs are macroversion 7.0/6.1 compatible with a shared-memory interface. The DACs can support one or two 16-Mbit or one 64-Mbit 125-MHz SDRAM. This chip is integrated with UARTs, SmartCards, I²C controller, and 3 PWM outputs, modem support, and a 44-bit programmable I/O.

**FIGURE 21.25**

Single-chip decoder by STMicroelectronics.

The transport Demux includes a DES and DVB descrambler and 32 PID support, and it can handle serial and parallel input. The block diagram is shown in Fig. 21.25.

STMicroelectronics produced the STi7000 HD MPEG-2 video decoder and display in 2000. This chip is a real-time high-definition MPEG-1 and -2 video decoder. The video rates are $1920 \times 1088 \times 30$ Hz interlaced or $1280 \times 720 \times 60$ Hz progressive. Memory reduction schemes are used to restrict the memory requirements to 64 Mbits for hard drive applications. Video format converters are integrated and an MPEG-2 MP@HL-compliant video decoder is used. OSD with 1, 2, 4, or 16 bits/pixel and full, one-half, or one-third resolution is supported. The output can be either 8-bit, 27-MHz CCIR 601 compatible or 24-bit, 81-MHz 4:4:4 output. D1 video input and a standard 8-bit interface for microcontrollers and compressed data inputs are used. A clock speed of 27 MHz is required for the entire chip. The chip uses $0.35\text{-}\mu\text{m}$ CMOS technology with 3.3 V.

STMicroelectronics released the STi4600 Dolby Ac-3 Audio decoder in 2000. This chip decodes 5.1 and 2 channels with DVD standards. This device also decodes MPEG layers I and II with a precision of 24 bits. The input can be a serial or parallel MPEG-2 PES stream. The controller interface can be I²C or 8-bit parallel. External memory is not required if the delay requirement is higher than 35 ms. The output can be up to six channels. The PCM mode is transparent down-sampling 96 to 48 KHz. Prologic decoders are used. This chip uses a 3.3-V supply voltage.

The other products related to digital video chips are STi5500, STi5505, STi5512, and ST20TP4. STi55XX is a set-top/DVD backend decoder with host processors. STi5512 is a tuned broadcast application and STiT20TP4 is a programmable transport IC for broadcast and DirecTV applications.

21.4.2.8 Single-Chip Decoder by Zoran

Zoran Inc. released the ZR36215 single-chip audio/video decoder. This chip supports SuperVCD standards along with Video CD1.1/2.0. The product also supports CD-DA. This chip has a video decoder that decodes MPEG-2 and MPEG-1 with a variable bit-rate. The MPEG-2 audio decoder uses a sampling rate of 44.1 or 48 KHz. An audio decoder with MPEG-2 provides multiple channels and is backward compatible with MPEG-1. MPEG-1 standards are used for VideoCD. The outputs can be mono or stereo. This chip supports multiple features of Karaoke. The programmable 40-MIPS DSP can process various audio algorithms to enhance audio effects in a two-speaker environment (Fig. 21.26).

Zoran produced the ZR38601 Programmable Digital Audio Processor. This product is the fourth-generation decoder made by Zoran that handles real-time Dolby Digital 5.1-channel and MPEG-2 digital surrounding algorithms. The floating-point hardware used in this product is optimal for audio applications. With a single DAC, an optical interface for the S/PDIF input, and an oscillator crystal, the chip produces standard decoding functions. It may have 8 channels of output, analog inputs, and long delay memories. The audio decoder handles a maximum bit-rate of 640 Kbits/s and can also use Dolby Pro-logic encoding and decoding. The decoder parses PES streams, decodes PTS formats, and handles SCR. The silicon software allows a magnitude of audio enhancements. The inputs can be serial or parallel streams. Serial SPI, serial Z2C, or 8-bit host interface is supported. Formatted input in the S/PDIF receiver can have a maximum 96-KHz sample rate. The output is formatted as S/PDIF Dolby Digital and MPEG transmitter. The chip has separate PLLs for DSP core and audio I/O. No external memory is needed for basic decoding. The chip operates at 3.3 V. The block diagram of ZR38601 is shown in Fig. 21.27.

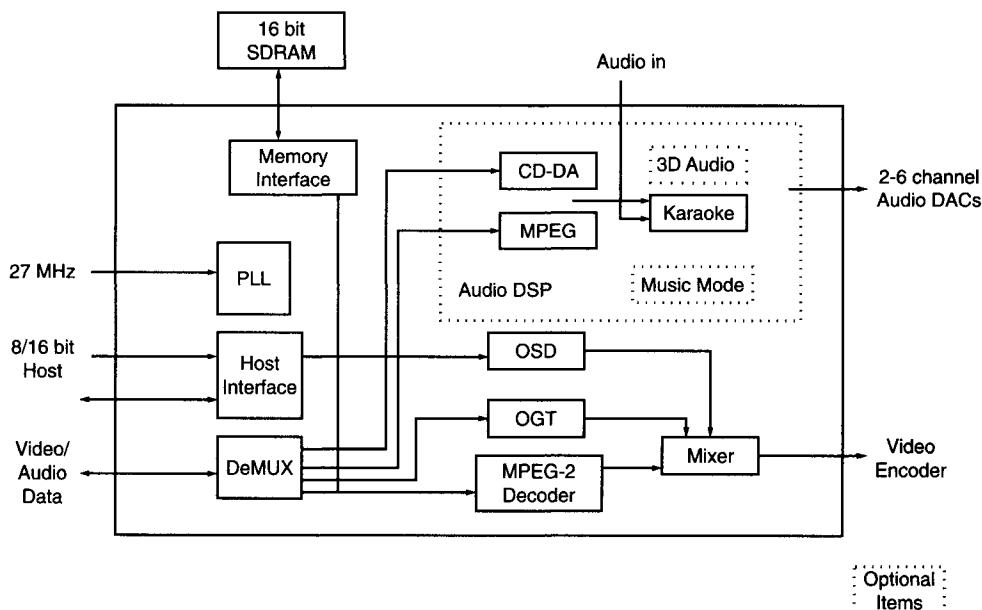
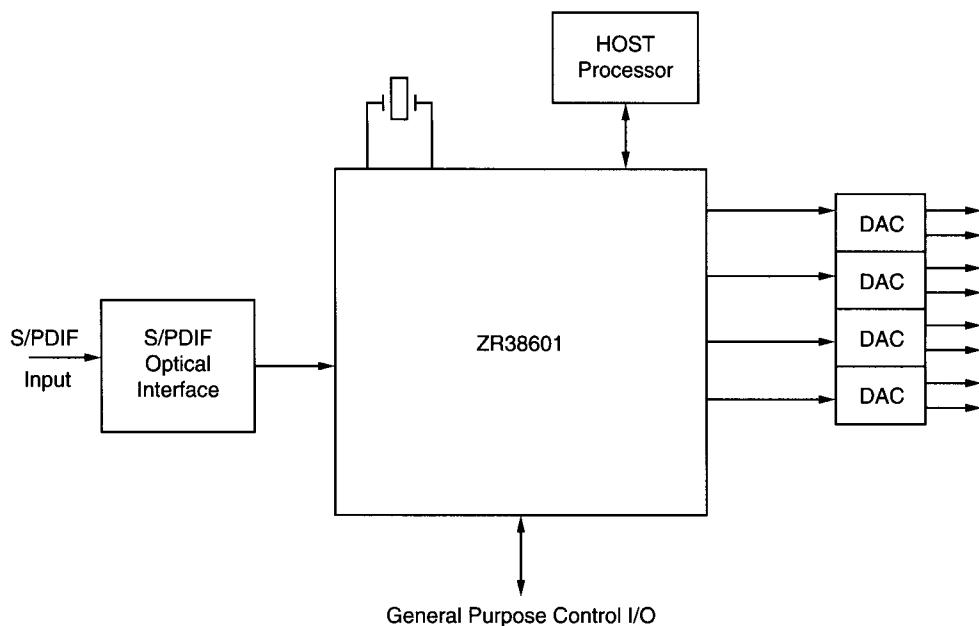


FIGURE 21.26

Single-chip decoder solution by Zoran for SuperVCD applications.

**FIGURE 21.27**

Single-chip programmable digital audio processor by Zoran.

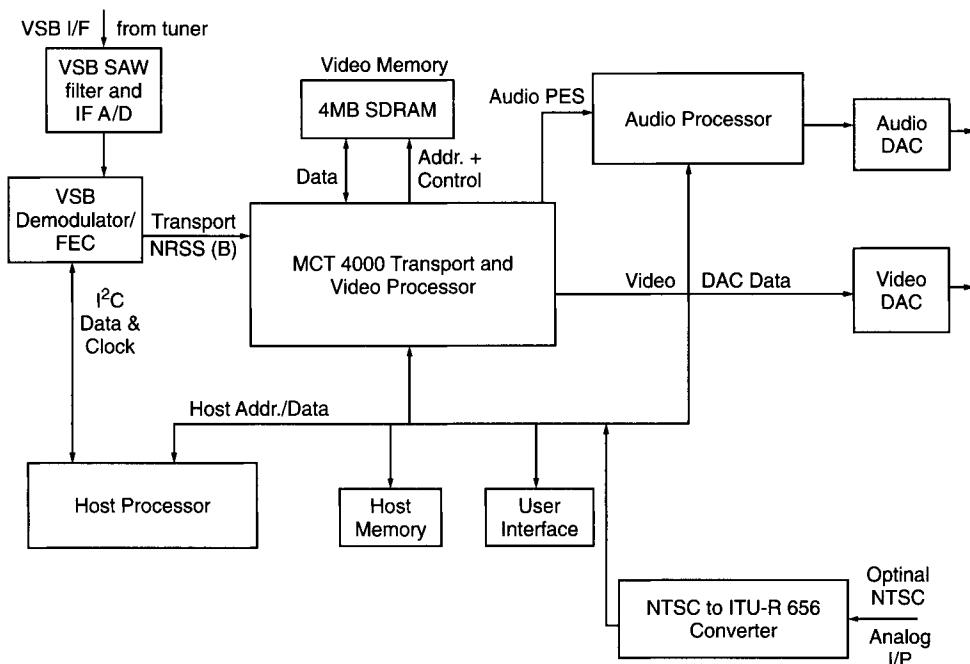
21.4.2.9 MPEG-2 Video Decoder by Motorola

Motorola developed MCT4000 as a transport and video processor for digital television applications following the Advanced Television System Committee (ATSC) standard. This device provides the MPEG-2 transport parsing function. The decoder accepts MPEG-2 MP@HL PES. The ATSC standards are followed as a guideline on whether to use 480 lines \times 720 pixels as interlaced or progressive outputs. This device is useful for set-top applications or television applications. This decoder requires only 4 Mb external SDRAM for the compression algorithm and postfiltering without compromising on good picture quality. Memory access is also optimized to have improved performance. The output end, OSD, is implemented with 256 colors, 64 levels of transparency, and 30-bit resolution. The chip integrates a high-performance video encoder that creates standard definition video output, NTSC, composite video, S-VHS, and RGB formats. The chip also has a bidirectional ITU-R 656 interface with other video processors and for reception of externally digitized analog video data. The chip can operate at 1.8 and 3.3 V for I/O. The chip provides an interface for audio processors. The block diagram is shown in Fig. 21.28.

21.4.2.10 MPEG-2 Digital Audio/Video Decoder by IBM

The MPEGGC22 audio/video decoder is a single chip that can decode MPEG-2 MP@ML video and MPEG-2 stereo layers I and II (CD quality) audio, released by IBM in 1997.

The video decoder handles MPEG-2 MP@ML for a 4:2:0 chroma application. The chip is compatible with European DVB standards. The inputs can be PES layers, Horizontal and vertical filters are used to deliver high-quality video. The chip delivers 11-bit DC precision for enhanced picture quality. Error concealment is performed at all video layers. The chip supports teletext or vertical blanking interval data support. The system provides 2 MB of SDRAM with a 16-bit interface for MPEG-2, NTSC, and PAL up to 15 Mbps.

**FIGURE 21.28**

Transport demultiplexer and video decoder by Motorola.

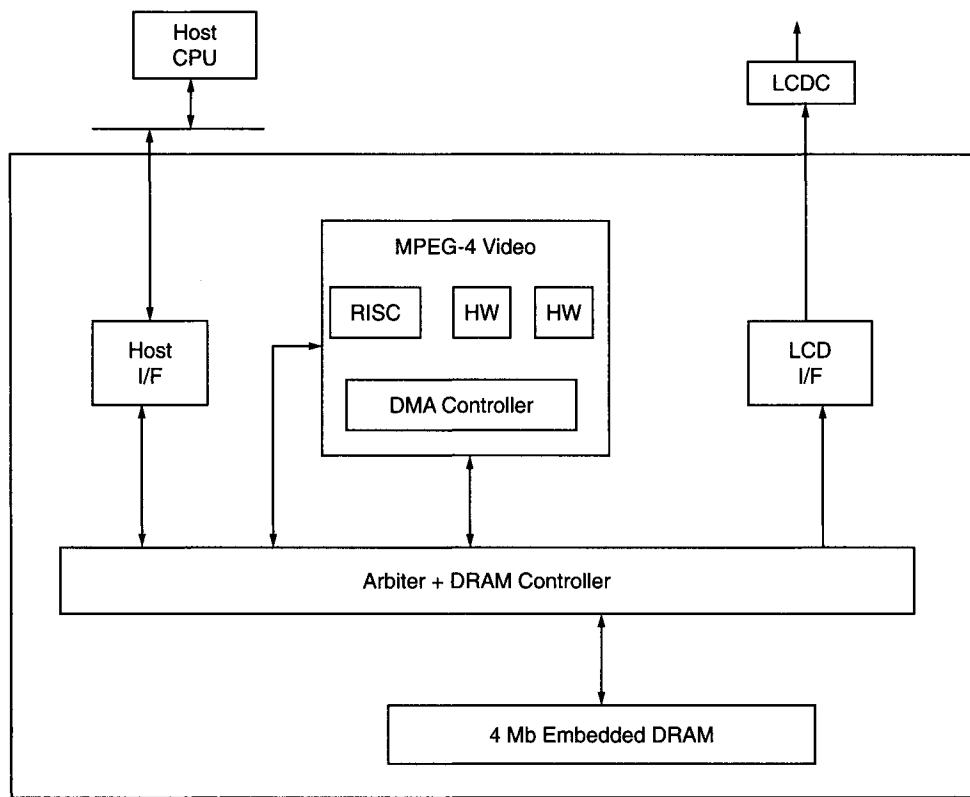
The audio decoder supports MPEG-2 layer I and layer II stereo. The decoder can accept PCM audio data. Sampling rates are 16, 22.05, 24, 32, 44.1, and 48 KHz. Audio is attenuated at 63 increments per channel. The chip operates at 27 MHz. The OSD is programmable, and video shading and blending can be performed. The input of 8- or 16-bit audio/video compression data and host interface is used. Serial input data can also be provided for the transport and host interface. The chip is built on 0.4- μ m CMOS technology with 3.3 V and it consumes 1.4 W of power.

21.4.2.11 MPEG-4 Products by Toshiba

Toshiba developed the LSI TC35274 MPEG-4 video decoder in 2000. The decoder chip performs at 15 frames/s. The MPEG-4 video is decoded with QCIF (Quarter Common Intermediate Format) (176×144 pixels) and the clock is 30 MHz. The chip includes a 4-bit embedded DRAM for low power consumption. The core of the decoder is a 16-bit RISC processor and dedicated hardware accelerator. The firmware program for the RISC to run the chip as an LSI decoder is downloaded into the DRAM before starting. The MPEG standard is ISO MPEG-4 SP@L1 (simple profile at level 1). The output is 4:2:2 Y:Cb:Cr 8-bit digital image data for LCD. The block diagram is shown in Fig. 21.29.

Toshiba also released the MPEG-4 audiovisual codec LSI in 2001. The TC35273 chip supports 3GPP 3G-342M video telephony systems with QCIF format and 15 frames/s. Adaptive multirate speech codec and ITU-T h.223 are executed concurrently at 70 MHz. The unit contains video, audio, and a multiplexer unit. A DRAM of 12 Mbits is shared by all three units. The video codec, audio codec, and multiplexer are each 16-bit RISC processors. All the units are integrated and synchronized by PLLs.

The video codec supporting the MPEG standard is ISO MPEG-4 SP@L1 with QCIF. The input is 4:2:2 Y:Cb:Cr 8-bit digital image data. A CMOS camera or an NTSC decoder can be connected. The output can be 4:2:2 Y:Cb:Cr 8-bit digital image data for LCD, or an NTSC

**FIGURE 21.29**

MPEG-4 video decoder LSI by Toshiba.

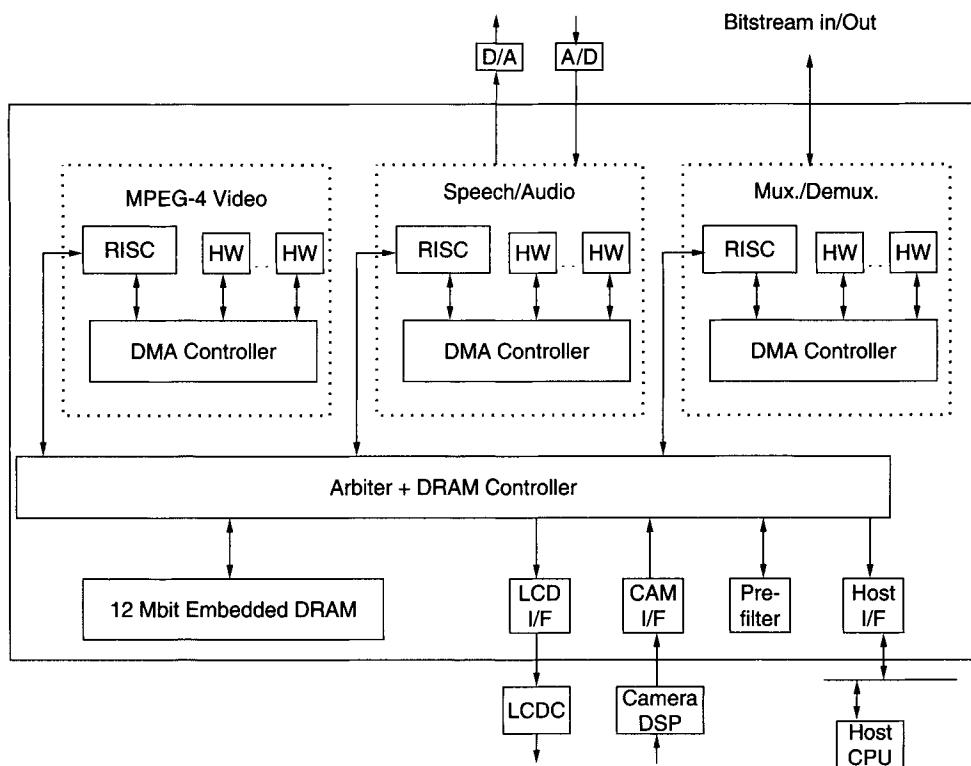
encoder is connected. The audio codec uses the AMR speech codec at 8 Kbps with CS-ACELP or the ITU-T G.729 speech codec at 8 Kbps with Cs-ACELP, and other formats. The stereo audio decoder runs at 96 Kbps with a maximum frequency of 44.1 KHz. The decoder can also handle the ISO/IEC 13818-7 AAC LC audio decoder at 144 Kbps with a maximum of 48 KHz of sampling frequency. The input and output can be in PCM format. The multiplexer and demultiplexing units operate at 32 and 384 Kbps, respectively. The unit can handle bitstream input/output via a serial interface. The block diagram is seen in Fig. 21.30.

21.4.2.12 MPEG-4 Decoder by Sigma Designs Inc.

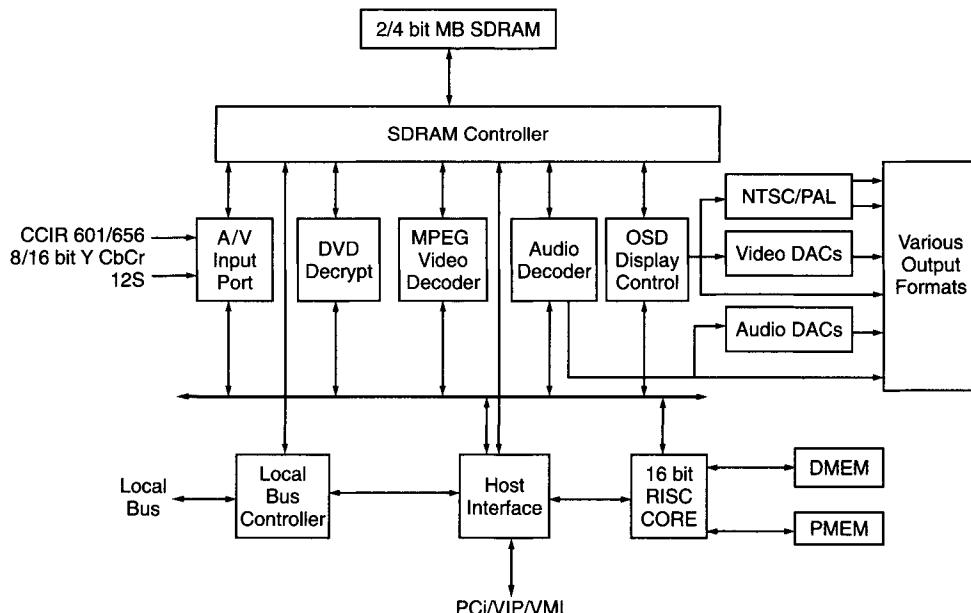
Sigma Design Inc. developed the EM8470 MPEG-1, MPEG-2 MP@ML, and MPEG-4 SP@L1 decoders for set-top DVD. This chip is based on the REALmagic video streaming technology. The formats that the product handles are DVD-video, DVD-audio, SVCD, VCD, CD-DA, and CD-ROM. The audio decoding is performed according to Dolby Digital AC-3 (two channels), MPEG-1 (layers I and II, down-mixed to two channels), and DVD-audio linear PCM. The output can be interlaced or progressive. The block diagram is shown in Fig. 21.31.

21.4.2.13 Other Chip-sets

We describe some other industry products in this section, just highlighting the dominant features of each of them. We mention products released by Sigma Design Inc. Cirrus Logic, Jacob Pineda Inc., MICRONAS, Philips, Conexant, Analog Devices, Amlogic, and Acer Laboratories. In Tables 21.1, 21.2, 21.3, and 21.4, the features are presented in tabular form. We find that most of the applications are DVD compliant and some are compliant with the broadcast standard.

**FIGURE 21.30**

MPEG-4 audiovisual codec LSI by Toshiba.

**FIGURE 21.31**

MPEG-4 video decoder by Sigma Designs Inc.

Table 21.1 Available MPEG Audio or Video Compression Products from Other Vendors

Product	Company	Year	Main Purpose	Video or Audio Standard
Audio products				
CS492301	Cirrus Logic	1999	Audio decoder	Dolby Digital, MPEG-2, PCM
J1	Jacob Pineda Inc.	1997	Audio decoder	5.1-Channel Dolby, AC-3, MPEG layers I and II
MAS3507D	MICRONAS	1998	Audio decoder (low power)	MPEG-1 and -2 layers 2 and 3 (for compression up to 12:1)
MAS3509F	MICRONAS	2000	Audio stereo decoder (low power)	MPEG2-AAC layers 2 and 3
SAA2502	Phillips	1997	Audio source decoder	MPEG-1 layers 1 and 2 and MPEG-2
Video products				
EM8220	Sigma Designs Inc.	1999	MPEG-2/DVD decoder	MP@ML MPEG-2, MPEG-1
VideoFLOW	Array Microsystems	1997	Real-time video encoder	IBBP or I-frame MPEG-1 encoding

Table 21.2 Available MPEG Audio or Video Compression Products from Other Vendors

Product	Frequency (MHz)	Throughput	O/P Format	Host CPU
Audio products				
CS492301	12.288/27	500 Kbps/1.5 Mbps	I2S/left,right justified	24-bit DSP
J1	54	640 Kbps	16-bit PCM, S/PDIF	Commercial
MAS3507D	14.725	<64 Kbps	Multiple formats	RISC CPU
MAS3509F	13–28	<64 Kbps	PCM, S/PDIF	RISC CPU
SAA2502	24.576	<417.96 Kbps	I2S, S/PDIF, 256 or more over sampled analog audio	
Video products				
EM8220	27	64 Kbps	YUV format: CCIR supported resolution for NTSC and PAL	66 MIPS RISC
VideoFLOW	Not available	64 Kbps/3.07 Mbps	MPEG-1	Not available

Table 21.3 Available MPEG Combined Video and Audio Compression Products from Other Vendors

Product	Company	Year	Main Purpose	Video Decoding Standard	Audio Decoding Standard
REALmagic DVR	Sigma Designs Inc.	2000	Digital video recorder	MPEG-1, MPEG-2	MPEG-1 layers 1 and 2
CS98000	Cirrus Logic	2000	Internet DVD	MPEG for DVD, VCD, SVCD	5.1-channel AC-3, MPEG stereo
CX22490/CX22491	Conexant	2000	MPEG video processor	MP@ML, MPEG-2	Dolby Digital, MPEG-1 and MPEG-2
AML3250	Amlogic Inc.	2000	DVD application	MPEG-2, MP@ML	Dolby Ac-3 5.1 channel
M3321	Acer Laboratories	1997	DVD and STB application	MPEG-1, MPEG-2, MP@ML	MPEG-1, MP3, Dolby Digital, DVD LPCM, MPEG-2

Table 21.4 Available MPEG Combined Video Audio Compression Products from Other Vendors

Product	Frequency (MHz)	Throughput	O/P Format	Host CPU
REALmagic DVR	266	500 Kbps to 15 Mbps	NTSC, PAL, MPEG-1, MPEG-2, PCM/WAV	Pentium
CS98000	27	Not available	8-channel PCM output,	Dual 32-bit RISC, 32-bit DSP (A)
CX22490/CX22491	160	Not available	NTSC, PAL with CCIR-supported resolution	32-bit RISC
AM3250	Not available	Not available	NTSC, PAL, CCIR, S/PDIF(A), PCM(A)	RMRISC 32-bit (A)
M3321	27	Not available	I2S(A), CCIR, PAL	Not available

21.4.2.14 Commercial Web Pages for the Vendors

Acer Laboratories at <http://www.ali.com.tw/>
Amlogic Inc. at <http://www.amlogic.com>
Analog Devices at <http://www.analog.com/>
Array Microsystems at <http://www.array.com/>
ATMEL at <http://www.atmel.com>
C-Cube at http://www.c_cube.com
Cirrus Logic at <http://www.cirrus.com>
Conexant Inc. at <http://www.conexant.com>
Fujitsu at <http://www.fujitsu.org>
IBM Inc. at <http://www.ibm.com>
Intel at <http://www.intel.com>
Jacob Pineda Inc. at <http://www.jacobspineda.com>
LSI Logic at <http://www.lsilogic.com/index4.html>
Lucent Technologies at <http://www.lucent.com/micro>
MICRONAS at <http://www.micronas.com>
Motorola Inc. at <http://www.motorola.com>
NEC Electron Devices at <http://www.nec.com>
Philips Semiconductors at <http://www-eu2.semiconductors.philips.com/cms/>
Sigma Designs Inc. at <http://www.sigmadesigns.com>
STMicroelectronics at <http://us.st.com>
Texas Instruments at <http://www.ti.com>
Toshiba America Inc. at <http://www.toshiba.com>
Zoran at <http://www.zoran.com>

21.5 REFERENCES

1. LSI Logic, 64700 Series Data-Sheet.
2. Kuroda, T., 1997. A 0.9 V, 150 MHz, 10 mW, 4 mm 2,2-D discrete cosine transform core processor with variable-threshold-voltage scheme. In *Digest of IEEE International Solid-State Circuits Conference*, pp. 166–167, February 1997.
3. Artieri, A., E. Macoviak, F. Jutand, and N. Demassieux, 1988. A VLSI one chip for real time two-dimensional discrete cosine transform. In *Proceedings of International Symposium on Circuits and Systems*.
4. Mukherjee, A., N. Ranganathan, J. Fleider, and T. Acharya, 1993. MARVLE: A VLSI chip for data compression using tree-based codes. *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 203–214, June 1993.
5. Mukherjee, A., N. Ranganathan, and M. Bassiouni, 1991. Efficient VLSI designs for data transformation of tree-based codes. *IEEE Transactions on Circuits and Systems*, Vol. 38, No. 3, pp. 306–314, March 1991.
6. Mukherjee, A., N. Ranganathan, and M.A. Bassiouni, 1989. Adaptive and pipelined VLSI designs for tree-based codes. In *Proceedings of the 1989 IEEE International Conference on Computer Design*, pp. 369–372.
7. Wu, A., and K.J.R. Liu, 1998. Algorithm-based low-power transform coding architectures: The multirate approach. *IEEE Transactions on VLSI Systems*, Vol. 6, No. 4, pp. 707–717, December 1998.
8. Lewis, A.S., and G. Knowels, VLSI Architecture for 2-D Daubechies Wavelet Transform without Multipliers.
9. Ackland, B., 1993. Video compression and VLSI. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 11.1.1–11.1.6.

10. Jung, B., and W. Burleson, 1994. A VLSI systolic array architecture for Lempel-Ziv based data compression. In *Proceedings of 1994 IEEE International Symposium on Circuits and Systems*, Vol. 3, pp. 65–68.
11. Jung, B., and W.P. Burleson, 1998. Efficient VLSI for Lempel-Ziv compression in wireless data communication networks. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, Vol. 6, No. 3, pp. 475–483, September 1998.
12. Tseng, B.D., and W.C. Miller, 1978. On computing discrete cosine transform. *IEEE Transactions on Computers*, Vol. C-27, No. 10, pp. 966–968.
13. Lee, B.G., 1984. A new algorithm to compute the discrete cosine transform. In *Proceedings of International Conference on Acoustics, Speech and Signal Process*, Vol. ASSP, pp. 1243–1245, December 1984.
14. Cafforio, C., and F. Rocca, 1976. Methods for measuring small displacements of television images. *IEEE Transactions on Information Theory*, Vol. IT-22, pp. 573–579, September 1976.
15. Chakrabarti, C., and M. Vishwanath, 1995. Architectures for wavelet transforms: A survey. *Journal of VLSI Signal Processing*, Vol. 10, pp. 225–236.
16. Chakrabarti, C., and M. Vishwanath, 1995. Efficient realization of the discrete and continuous wavelet transforms: From single chip implementations to mappings on SIMD array computers. *IEEE Transactions on Signal Processing*, Vol. 43, No. 3, pp. 759–771, March 1995.
17. Chakrabarti, C., and M. Vishwanath, 1996. Architectures for wavelet transforms: A survey. *Journal of VLSI Signal Processing*, Vol. 14, pp. 171–192.
18. C-Cube Microsystems, 1992. *CL550 Users Manual*.
19. Lin, C.-H., C.-M. Chen, and C.-W. Jen, 1996. Low power design for MPEG-2 video decoder. *IEEE Transactions on Consumer Electronics*, Vol. 42, No. 3, pp. 513–521, August 1996.
20. Chen, C.-T., and L.-G. Cheni, 1997. High-speed VLSI design of the LZ-based data compression. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, Vol. 3, pp. 2064–2067.
21. Huffman, D., 1952. A method for the construction of minimum redundancy codes. *Proceedings of IRE*, Vol. 40, pp. 1098–1101.
22. Johnson, D., V. Akella, and B. Scott, Micropipelined Asynchronous Discrete Cosine Transfer (DCT/IDCT) Processor.
23. Milovanovic, D., and Z. Bojkovic, 1999. MPEG-4 video transmission over the Internet. In *Proceedings of IEEE Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services*, pp. 309–312, June 1999.
24. Wu, D., Y.T. Hou, W. Zhu, H.-J. Lee, T. Chiang, Y.-Q. Zhang, and H.J. Chao, 2000. On end-to-end architecture for transporting MPEG-4 video over the Internet. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 10, No. 6, pp. 923–941, September 2000.
25. Royals, D.M., T. Markas, N. Kanopoulos, J.H. Reif, and J.A. Storer, 1993. On the design and implementation of a lossless data compression and decompression chip. *IEEE Journal of Solid-State Circuits*, Vol. 28, No. 9, pp. 948–953, September 1993.
26. Charot, F., G. L. Fol, P. Lemonnier, C. Wagner, and C. Bouville, 1999. Towards hardware building blocks for software-only real-time video processing: The MOVIE approach. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 9, No. 6, pp. 882–894, September 1999.
27. Langdon, G., 1984. An introduction to arithmetic coding. *IBM Journal of Research and Development*, Vol. 28, No. 2, pp. 135–149, March 1984.
28. Chang, H-C., Y-C. Chang, Y-B. Tsai, C-P. Fan, and L-G. Chen, 2000. MPEG-4 video bitstream structure analysis and its parsing architecture design. In *Proceedings of International Symposium on Circuits and Systems*, pp. 184–187.
29. Park, H., and V.K. Prasanna, 1993. Area efficient VLSI architectures for Huffman coding. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 40, No. 9, pp. 568–575, September 1993.
30. Sato, H., et al., 2000. MPEG-2 4:2:2@HL encoder chip set. In *Proceedings of International Symposium on Circuits and Systems*, Vol. 4, pp. 41–44.
31. Sun, H., W. Kwok, and J.W. Zdepski, 1996. Architectures for MPEG compressed bitstream scaling. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, pp. 191–199, April 1996.

32. Lee, H.-Y., L.-S. Lan, M.-H. Sheu, and C.-H. Wu, 1996. A parallel architecture for arithmetic coding and its VLSI implementation. In *Proceedings of the IEEE 39th Midwest Symposium on Circuits and Systems*, Vol. 3, pp. 1309–1312.
33. Yamauchi, H., 1992. Architecture and implementation of a highly parallel single chip video DSP. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, pp. 207–220, June 1992.
34. Chuang, H.Y.H., and L. Chen, 1995. VLSI architecture for fast 2-D discrete orthonormal wavelet transform. *Journal of VLSI Signal Processing*, Vol. 10, pp. 225–236.
35. Intel, 1993. *82750DB Display Processor Databook*, Santa Clara, CA, September 1993.
36. Intel, 1993. *82750PB Pixel Processor Databook*, Santa Clara, CA, October 1993.
37. Bae, J., and V.K. Prasanna, Synthesis of VLSI architecture for two-dimensional discrete wavelet transforms. In *Proceedings of IEEE International Conference on Application Specific Array Processors*, pp. 174–181, July 1995.
38. Biemond, J., L. Looijenga, and D. E. Boekee, 1987. A Pel-recursive Wiener-based displacement estimation algorithm for interframe image coding applications. In *Proceedings of SPIE Visual Communication and Image*, Vol. II-845, pp. 424–431.
39. Chen, J., and M.A. Bayoumi, 1995. A scalable systolic array architecture for 2-D discrete wavelet transforms. In *Proceedings of IEEE VLSI Signal Processing Workshop*, pp. 303–312.
40. Kneip, J., B. Schmale, and H. Moller, 1999. Applying and implementing the MPEG-4 multimedia standard. *IEEE Micro*, Vol. 19, No. 6, pp. 64–74, November/December 1999.
41. Limb, J. O., and J. A. Murphy, 1975. Measuring the speed of moving objects from television images. *IEEE Transactions on Communication*, Vol. COM-23, pp. 474–478, April 1975.
42. Venbrux, J., P.-S. Yeh, and M.N. Liu, 1992. A VLSI chip set for high speed lossless data compression. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 2, No. 4, pp. 381–391, December 1992.
43. Ziv, J., and A. Lempel, A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Vol. IT-23, No. 5, pp. 337–343.
44. Storer, J.A., and J.H. Reif, 1990. A Parallel architecture for high speed data compression. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pp. 238–243.
45. Jou, J.M., and P.-Y. Chen, 1999. A Fast and efficient lossless data-compression method. *IEEE Transactions on Communications*, Vol. 47, No. 9, pp. 1278–1283, September 1999.
46. Lin, K.-J., and C.-W. Wu, 2000. A low-power CAM design for LZ data compression. *IEEE Transactions on Computers*, Vol. 49, No. 10, pp. 1139–1145, October 2000.
47. Kim, A pipelined systolic array architecture for the hierarchical block-matching algorithm. In *Proceedings of International Symposium on Circuits and Systems*, Vol. 3, pp. 221–224.
48. Parhi, K.K., and T. Nishitani, VLSI architectures for discrete wavelet transforms. *IEEE Transactions on VLSI Systems*, Vol. 1, No. 2, pp. 191–202, June 1993.
49. Komarek, 1989. Array architectures for block matching algorithms. *IEEE Transactions on Circuits and Systems*, Vol. 36, October 1989.
50. Rao, K.R., and P. Yip, 1990. *Discrete Cosine Transform*. Academic Press, San Diego, CA.
51. De Vos, L., 1990. VLSI architectures for the hierarchical block matching algorithm for HDTV applications. In *Proceedings of SPIE Visual Communication and Image*, Vol. 1360, pp. 398–409.
52. Liu, L.-Y., J.-F. Wang, R.-J. Wang, and J.-Y. Lee, 1994. CAM-based VLSI architectures for dynamic huffman coding. *IEEE Transactions on Consumer Electronics*, Vol. 40, No. 3, pp. 282–289, August 1994.
53. LSI Logic, 1993. JPEG Chipset Technical Manual, Milpitas, CA.
54. LSI Logic, 1993. L64702 JPEG Coprocessor Technical Manual, Milpitas, CA.
55. Berekovic, M., P. Pirsch, T. Selinger, K.-I. Wels, A. Lafage, C. Heer, and G. Chigo, 2000. Architecture of an image rendering co-processor for MPEG-4 systems. In *Proceedings of ICASSAP*, pp. 15–24. IEEE Press, San Mateo, CA.
56. Gonzalez-Smith, M., and J. Storer, 1985. Parallel algorithms for data compression. *Journal of ACM*, Vol. 32, No. 2, pp. 344–373, April 1985.
57. Kovac, M., and N. Ranganathan, 1995. JAGUAR: A fully pipelined VLSI architecture for JPEG image compression method. *Proceedings of the IEEE*, Vol. 83, No. 5, pp. 247–258, February 1995.
58. Takahashi, M., 2000. A scalable MPEG-4 Video CODEC architecture for IMT-2000 multimedia applications. In *Proceedings of IEEE Symposium on Circuits and Systems*, pp. 188–191, May 2000.

59. Vishwanath, M., R.M. Owens, and M.J. Irwin, VLSI architectures for the discrete wavelet transform.
60. Liu, M.N., 1996. MPEG decoder architecture for embedded applications. *IEEE Transactions on Consumer Electronics*, Vol. 42, No. 4, pp. 1021–1028, November 1996.
61. Sun, M.T., 1991. VLSI architecture and implementation of a high speed entropy decoder. In *Proceedings of International Symposium on Circuits and Systems*, pp. 200–202.
62. Sun, M.T., L. Wu, and M.L. Liou, 1987. A concurrent architecture for VLSI implementation of discrete cosine transform. *IEEE Transactions on Circuits and Systems*, Vol. 34, No. 8, pp. 992–994, August 1987.
63. Demassieux, N., and F. Jutland, 1993. *Orthogonal Transform*, pp. 217–250. Elsevier Science, Amsterdam.
64. Ranganathan, N., and S. Henriques, 1993. High-speed VLSI designs for Lempel-Ziv based data compression. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, Vol. 40, No. 2, pp. 96–106, February 1993.
65. Wu, P-C., and L-G. Chen, 2001. An efficient VLSI architecture for two-dimensional discrete wavelet transform. *IEEE Transactions on CAS for Video Technology*, Vol. 11, No. 4, pp. 536–545, April 2001.
66. Pirsch, P., and H.J. Stolberg, 1997. Architectural approaches for video compression. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 176–185.
67. Pirsch, P., N. Demassieux, and W. Gehrke, 1995. VLSI architectures for video compression—A survey. *Proceedings of the IEEE*, Vol. 83, No. 2, pp. 220–246, February 1995.
68. Ruetz, P., and P. Tong, 1992. A 160-Mpixels/s IDCT processor for HDTV. *IEEE Micro*, Vol. 12, No. 5, pp. 28–32, October 1992.
69. Sriram, P., S. Sudharsanan, and A. Gulati, 2000. MPEG-2 video decompression on a multi-processing VLIW microprocessor. In *Proceedings of IEEE Conference on Consumer Electronics*, June 2000.
70. Jain, P.C., W. Schlenk, and M. Riegel, 1992. VLSI implementation of two-dimensional DCT processor in real time for video codec. *IEEE Transactions on Consumer Electronics*, Vol. 38, No. 2, pp. 537–545, August 1992.
71. Zito-Wolf, R., 1990. A systolic architecture for sliding-window data compression. In *Proceedings of IEEE Workshop on VLSI Signal Processing*, pp. 339–351, November 1990.
72. Rice, R.F., P.-S. Yeh, and W.H. Miller, 1991. Algorithms for a very high speed universal noiseless coding module. In JPL Publication 91-1, JPL Propulsion Laboratory, Pasadena, CA.
73. Zito-Wolf, R.J., 1990. A broadcast/reduce architecture for high-speed data compression. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pp. 174–181.
74. Haralick, R.M., 1976. A storage efficient way to implement the discrete cosine transform. *IEEE Transactions on Computers*, Vol. C-25, 764–765, July 1976.
75. Hwang, S.-A., and C.-W. Wu, 2001. A unified VLSI systolic array design for LZ data compression. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, Vol. 9, No. 4, pp. 489–499, August 2001.
76. Purcell, S.C., and D. Galbi, 1992. C-Cube MPEG video processor. In *Proceedings of SPIE Image Processing and Interchange*, Vol. 1659.
77. Golomb, S., 1996. Run-length encodings. *IEEE Transactions on Information Theory*, Vol. IT-12, pp. 399–401, July 1966.
78. Paek, S.-K., and L.-S Kim, 2000. A real-time wavelet vector quantization algorithm and its VLSI architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 10, No. 3, pp. 475–489, April 2000.
79. Chang, S.J., M.H. Lee, and J.Y. Park, 1997. A high speed VLSI architecture of discrete wavelet transform for MPEG-4. *IEEE Transactions on Consumer Electronics*, Vol. 43, No. 3, pp. 623–627, August 1997.
80. Kim, S.P., and D.K. Pan, 1992. Highly modular and concurrent 2-D DCT chip. In *Proceedings of International Symposium on Circuits and Systems*.
81. Winograd, S., 1978. On Computing the Discrete Fourier Transform.
82. Akari, T., 1994. Video DSP architecture for MPEG2 CODEC. In *Proceedings of ICASSP*, Vol. 2, pp. 417–420. IEEE Press, San Mateo, CA.

83. Fautier, T., 1994. VLSI implementation of MPEG decoders. In *Proceedings of International Symposium on Circuits and Systems*.
84. Inoue, T., 1993. A 300-MHz BiCMOS video signal processor. *IEEE Journal of Solid-State Circuits*, Vol. 28, December 1993.
85. Huang, T. S., 1983. *The Differential Method for Image Scene Analysis*. Springer-Verlag, Berlin/New York.
86. Totzek, U., F. Matthiesen, S. Wohllenben, and T.G. Noll, 1990. CMOS VLSI implementation of the 2D-DCT with linear processor arrays. In *Proceedings of International Conference on Acoustics, Speech and Signal Process*, Vol. 3.
87. Bhaskaran, V., and K. Konstantinides, 1997. *Image and Video Compression Standards: Algorithms and Architectures*, 2nd ed. Kluwer Academic, Dordrecht/Norwell, MA.
88. Badawy, W., and M. Bayoumi, 2000. VLSI architecture for hybrid object-based video motion estimation. In *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*, May 2000.
89. Gehrke, W., R. Hoffer, and P. Pirsch, 1994. A hierarchical multi-processor architecture based on heterogeneous processors for video coding applications. In *Proceedings of ICASSP*, Vol. 2. IEEE Press, San Mateo, CA.
90. Lai, Y.-K., and K.-C. Chen, 2000. A novel VLSI architecture for Lempel-Ziv based data compression. In *Proceedings on IEEE International Symposium on Circuits and Systems*, Vol. 5, pp. 617–620.
91. Tung, Y.-S., J.-L. Wu, and C.-C. Ho, 2000. Architecture design of an MPEG-4 system. In *Proceedings of International Conference on Consumer Electronics*, pp. 122–123.
92. Arai, Y., T. Agui, and M. Nakajima, 1998. A fast DCT-SQ scheme for images. *Transactions on IEICE*, Vol. E71, No. 11, pp. 1095–1097.

INDEX

A

- ACB algorithm, Buynovsky's, 201–203
Adaptive coding
binary arithmetic coding, 134–139
cumulative distribution update
 direct, 133–134
 periodic, 140–142
 tree-based, 139–140
strategies for computing symbol distributions, 132
Adaptive entropy coder
code selection, 317
fundamental sequence encoding, 313–314
low-entropy options
 second-extension option, 316
 zero-block option, 316–317
no compression, 317
split-sample option, 314–315
Adaptive Huffman coding
algorithm M , 94
brute force, 89–90
Faller, Gallager, and Knuth (FGK) algorithm, 91–93
splay tree algorithm, 93–94
Vitter's algorithm Λ , 93
Additive codes, 72–73, 77
Algorithmic Information Theory, *see* Kolmogorov complexity
Algorithmic Prefix Complexity, 42–43, 48
Algorithm M , 94
Amplitude range, and segmentation, 259–260
Apostolico and Fraenkel codes, 67–69, 77
Approximate arithmetic, 115–118
Arithmetic coding

- adaptive, 336
approximate arithmetic, 115–118
code values, 104–106
context-based, 397
correct decoding: conditions for, 118–120
decoding process, 110–111
dynamic sources, 112–113
encoder and decoder synchronized decisions, 113
encoding process, 106–109
entropy-coding methods, 102 intervals
 positive-length and disjoint, 118–119
 rescaling, 114–115
inverse operations, 120
model for Burrows–Wheeler compression, 177–179
nested subintervals, 119–120
notation, 103–104
optimality, 111–112
practical problems, 102–103
separation of coding and source modeling, 113–114
Arithmetic coding implementation, 120–147
adaptive coding, 132–142
coding with fixed-precision
 arithmetic, 121–132
complexity analysis, 142–147
integer-based, 147–150
Arithmetic encoder, 19–21
Arithmetic operations, speed of, 146–147
Asymptotic equipartition property, 26–34
ATMEL digital audio decoder, 432–433
Average filter, 380

B

- Backup, remote, massive data sets, 280
Balanced pairs, *rsync* algorithm, 283–284
Bandwidth, in telemetry compression, 250–251
Bayesian model averaging, 218
Bidirectional dictionary methods, 161
Bilevel image data, 336–338
Binary arithmetic coding, 134–139
Binary polynomial representation, 56, 74–75
Binary sources, 10
Biocompress programs, 162
Biomolecular sequences, compression, 162–163
Bisection search, 143–145
Bit-plane coding, JPEG2000, 360–364
Bit rate
 for lossless audio signal, 256
 variable, peak vs. average, 264
Bits per character
 averages, 240
 entropy and, 228–229
 for new transforms, 242
Bitstream scaling, architecture for, 421
Blending
 model-based, 218–219
 performance-based, 218
 probability domain, 219
Blocking system, 17–18
Block moves, in delta compression, 273
Block size, *rsync* algorithm, 281–283

- Block-sorting compression, *see* Burrows–Wheeler compression
- Bottom layer templates, 335–336
- Bottom layer typical prediction, 334
- Bounds
- error probability, 34
 - lower-bounding Kolmogorov complexity, 46–47
 - and redundancy and entropy, 16–17
- Brute force adaptive Huffman, 89–90
- Buffer carries, implementation with, 122–126
- Burrows–Wheeler algorithm, 170
- Burrows–Wheeler compression, 73–74, 77
- improvements to, 180–181, 190
 - Move-to-Front step, 183–185
 - elimination, 187–189
 - permutation, 181–183
 - preprocessing, 181
 - statistical compressor, 185–187
 - and symbol-ranking, 200
- Burrows–Wheeler transform
- in file synchronization, 189
 - final coder, 190
 - forward, 170–171
 - implementation, 173–174
 - for lossless algorithms, 233
 - for PNG, 374
 - relationship to other algorithms, 180
 - reverse, 171
 - algorithms for, 172–173
- Buynovsky’s ACB algorithm, 201–203
- Bzip2 compression algorithm, 234, 236–241
- C**
- CALIC, 301
- approach to modeling prediction error, 212
 - edge detection by, 216
- Canonical codes, Huffman, 84
- Carries
- buffer, implementation with, 122–126
 - propagation, 130–131
- Cascaded coding model, 178
- CCSDS, e_Rice algorithm recommendation, 312
- C-Cube video products, 427–428
- Chain rule, 7
- Channel coding, 251
- Checkpointing, 278
- Checksums, blockwise, of current file, 280–281
- Chip-sets, 438
- Church–Turing thesis, 38–39
- Codebook
- computed for classes only, 87
 - Huffman, 80–83
- Coded data format, 321
- Coder/decoder pairs, Huffman, 94–96
- Code selection function, adaptive entropy coder, 317
- Codestream, JPEG2000, 365–366
- Code values
- arithmetic coding, 104–111
 - scaling of, 115
- Codewords
- descendants, 14
 - end-of-line, 398
 - groups, in Golomb code, 62–64
 - ideal length, 16–17
 - in prefix code, 13–14
 - in tree, 15
 - variable-length, 12
- Coding distribution, 19
- Combined predictors, 217–219
- model-based blending, 218–219
 - performance-based blending of predictors, 218
 - probability domain blending, 219
- Comma codes
- adaptive entropy coder, 313–314
 - ternary, 70–71
- Competitive optimality, 25–26
- Complementary palindromes, 162
- Complete tree, 93
- Component transforms, JPEG2000, 354–355
- Compress anything claim, 43–44
- Compressed bitstream scaling, architecture for, 421
- Compressed domain image processing, JPEG2000, 353
- Compressed pointer macro scheme, 157
- Compression
- algorithms, 208
 - Burrows–Wheeler, *see* Burrows–Wheeler compression
 - data, *see* Data compression
 - dynamic Markov, 232
 - practical tips, 383–385
 - sequence, 20–21
 - symbol-ranking, 195–201
 - telemetry, *see* Telemetry compression tests, 385–388
- text, *see* Text compression
- Unicode, 293–294
- Compression engine, PNG, 374–376
- compress program, 165
- Computation
- issues, of Kolmogorov complexity, 44–47
 - sequential, of interval, 21–23
 - symbol distributions, strategies for, 132
- Computational complexity analysis
- arithmetic operations, 146–147
 - cumulative distribution estimation, 145–146
 - interval renormalization, 142–143
 - symbol search, 143–145
- Concrete Kolmogorov complexity, 40
- Consultative Committee for Space Data Systems, *see* CCSDS
- Content progressive representations, 340
- Context adaptive lossless image compression algorithm, *see* CALIC
- Correcting one-pass algorithm, 278
- Counting argument, 41
- Cumulative distribution
- of code values, 105–106
 - estimation, 145–146
 - updates
 - direct, 133–134
 - periodic, 140–142
 - tree-based, 139–140
- Current context, 202
- Current file, 270–271
- blockwise checksums of, 280–281
 - encoding, 274
- Cyclic redundancy code, 373
- D**
- Database records, and file systems: reconciling, 286–287
- Data compression
- audio, 255–267
 - different approaches, 35–36
 - lossless, CCSDS recommendation for space applications, 311–326
- Data compression: dictionary-based
- benchmark programs and standards, 165–166
 - biomolecular sequence compression, 162–163
 - data structures
 - Karp–Rabin fingerprints, 164–165

- suffix trees, 163–164
 trie-reverse trie pairs, 164
 tries and compact tries, 163
- dictionary construction, 154–161
 stages, 153
- Data structure
 and decoding control: JBIG2, 346–348
 dictionary-based data compression, 163–165
 JBIG, 336–338
 telemetry frame or packet, 251
- DCT, *see* Discrete Cosine Transform
- Decoding
 combined with updating and coding, 145
 correct: conditions for, 118–120
 Golomb code, 63–64
 JBIG, 338–339
 JPEG-LS, 309
 postprocessor and adaptive entropy decoder for, 321–323
 process, in arithmetic coding, 110–111
- Decoding: JBIG2
 generic refinement region, 342–343
 generic region, 341–342
 halftone region, 346
 pattern dictionary, 345–346
 rest region, 345
 symbol dictionary, 343–345
- Decoding control, and data structures: JBIG2, 346–348
- Decompression program
 custom-built, 41–42
 lossless audio data compression, 264–265
- Deflate, core of PNG’s compression scheme, 374–376
- Delta compression
 applications, 271–273
 block moves, 273
 choosing reference files, 278–279
 experimental results, 275–277
 LZ77-based, 274–275
 problem definition, 270–271
 space-constrained, 277–278
- Descriptional language, Kolmogorov complexity, 39–40
- Design decisions, PNG, 372–374
- Deterministic prediction, 333–334
- Dictionary
 for Golomb code, 62
 methods for lossless algorithms, 232–233
 pattern, decoding, 345–346
- static vs. dynamic construction, 154–161
 symbol, decoding, 343–345
- Dictionary-based data compression
 benchmark programs and standards, 165–166
 compressing biomolecular sequences, 162–163
 data structures, 163–165
 dictionary construction
 parsing issues, 155–157
 semidynamic and dynamic, 157–161
 static methods, 154–155
 stages, 153
- Differential compression, *see* Delta compression
- Differential layer prediction, 332–334
- Differential layer templates, 334
- diff* utility, 272
- Digit vector, combined with weight vector, 56–57
- Discrete Cosine Transform
 hardware, 416
 video compression, 419–420
- Distance coding, 233
- Distance measures, *rsync* algorithm, 283
 results, 285
- Distribution scheme, efficient, 270
- Dynamical dictionary methods, 157–161
 bidirectional, 161
 unidirectional, 158–161
- Dynamic Markov compression, 232
- Dynamical sources, arithmetic coding, 112–113
- E**
- e-commerce, security, 36
- Edge detection, by CALIC, 216
- Efficiency
 entropy coding, 208
 Huffman codes, 86
 implementation of Huffman codecs, 94–96
 memory-efficient algorithms, 95
 output, and renormalization, 127–130
 Rice codes, 61, 64
 speed-efficient algorithms, 95–96
- Elias algorithm, 19, 22–25
- Elias gamma codes, 58–59, 77
- Elias omega codes, 59
- emacs* data set, 275–276
- Encoding
 current file, 274
- fundamental sequence, 313–314
- JPEG-LS, 302–309
- prefix-free, 42–44
- process, in arithmetic coding, 106–109
- of ranks, 196
- run-length color, 398
- Encoding: JBIG
 adaptive arithmetic coding, 336
 bottom layer typical prediction, 334
 differential layer prediction, 332–334
 model templates, 334–336
 resolution reduction, 330–331
- Entropy
 achieving, 23–24
 adaptive entropy coder, 313–317
 and bits per character, 228–229
 conditional, 6
 interpretation, 8–9
 properties, 7–8
 definition, 3–4
 as information measure, 5–6
 joint, 6
 properties, 7
 properties, 4–5
- Entropy coding
 in compression systems, 102
 efficiency, 208
 JPEG-LS, 307–309
 lossless audio data compression, 262–263
- Entropy rate, 11
- Equivalence classes, Huffman code for, 87
- e_Rice algorithm, 312–313
- Error accumulation, 120
- Error modeling, in lossless image compression, 212
- Error probability
 bounded, 34
 and rate, 28
 optimal balance between, 31–33
- Escape character, 231
- Even–Rodeh code, 59–60
- Exclusion list, 198
- Explicit detection, predictors based on, 215–216
- Extended Huffman codes, 87–89
- External pointer macro scheme, 157
- F**
- Facsimile compression
 algorithms
 context-based arithmetic coding, 397
 modified Huffman, 393

Facsimile compression (*cont.*)
modified READ, 393–397
run-length color encoding, 398
historical overview, 391–393
standards
ITU-T Group 3 (T.4), 398–399
ITU-T Group 4 (T.6), 399
JBIG and JBIG2 (T.82 and T.88), 399
MRC-T.44, 399–402
Facsimile transmission, 87
JBIG compression for, 327–328
Faller, Gallager, and Knuth (FGK) algorithm, 91–93
Fibonacci codes
Apostolico and Fraenkel codes, 67–69
Fraenkel and Klein codes, 66
higher-order Fibonacci representation, 66–67
new order-3, 69–70
use as universal codes, 65
Zeckendorf representation, 66
Fibonacci polynomial representation, 57
Fictitious pixel, LNTP, 332–333
File synchronization
Burrows–Wheeler transform in, 189
remote, *see* Remote file synchronization
File systems, and database records: reconciling, 286–287
Filters, PNG, 378–383
Final coder, simplified, 190
Fingerprints, Karp–Rabin, 164–165
Fixed-length codes, for memoryless sources, 26–34
Fixed-precision arithmetic, coding with, 121–132
Flexible greedy parsing, 156
Formatting, JBIG, 336–338
Fraenkel and Klein codes, 66
Frame telemetry, 248
Freeze heuristics, 161
Frequency decay method, 91
Fujitsu, audio video products, 431–432
Fundamental sequence encoding, 313–314

G

Gamma codes, Levenshtein and Elias, 58–59
gcc data set, 275–276
GenCompress program, 162–163
GIF compression standard, 166, 372–373, 385–386

Gödel’s incompleteness theorem, 36–37
Goldbach G1 codes, 72, 77
Golomb codes, 62–64
limited-length, 307
Gradient adaptive predictor, 216
Grayscale image
GIF and PNG, 374
JBIG2, 339–340
Greedy strategy, of parsing, 156–157, 164
gzip compression program, 165, 233–234, 241, 276

H

Halftone region, decoding, 346
Hamming ball, 30–31
Hardware implementation
image compression, 415–417
text compression, 407–415
video compression, 417–442
Hierarchical lossless image coding, 220–222
Hierarchical prediction, in lossless image compression, 211–212
Hilbert scans, 213–214
Historical perspective
facsimile compression, 391–393
Kolmogorov complexity, 51
PNG, 371–372
symbol-ranking compressors, 197
unidirectional dictionary methods, 158
HTTP
performance improvement, 272
transfer, file synchronization for, 280
Huffman codes, 49–50
building of, 80–83
canonical, 84, 230
extended, 87–89
length-constrained, 89
modified, 87
N-ary, 83–84
performance, 84–86
prefixed, 87
Shannon–Fano coding, 80
Huffman coding
adaptive, 89–94
statistical method for lossless algorithms, 229–232
Huffman tree, 408–409
Human genome, 162

I

IBM, MPEG-2 digital audio/video decoder, 436–437

Image compression
bilevel, *see* JBIG; JBIG2
highly scalable, *see* JPEG2000
lossless, *see* Lossless image compression
Image compression hardware, 415–417
DCT, 416
JPEG, 417
wavelet architectures, 416–417
Image compression standard, GIF, 166
Image objects, 389
Image stripes, 328, 330
Implementations
arithmetic coding
adaptive coding, 132–142
complexity analysis, 142–147
with fixed-precision arithmetic, 121–132
with buffer carries, 122–126
Burrows–Wheeler compressor, 173–180
data compression for space applications, 324–326
and efficient output, 127–130
with integer arithmetic, 126–127
integer-based, 147–150
Lempel–Ziv methods, 161
numerical, lossless audio data compression, 263
Implementations: hardware
image compression, 415–417
text compression, 407–415
video compression, 417–442
Incompressibility, in Kolmogorov complexity, 40–42
Independent and identically distributed source, 10, 48, 102
Independent updating, 145–146
Inequality
frequently used in information theory, 4–5
Kraft–McMillan, 84–85
Kraft’s
in competitive optimality, 26
for prefix code, 14–17
Information measure
entropy as, 5–6
universal, 50–51
Information sources
discrete, 9
discrete stationary, 10–11
with memory, variable-length codes for, 18–26
memoryless
fixed-length codes for, 26–34
variable-length codes for, 11–18

- memoryless or i.d.d., 10
I
 Information theory, background, 228–229
 Initial letter preserving transform, 241–243
 Innovation entropy, 11
 Instantaneous code, 13
 Integer arithmetic, implementation with, 126–127, 147–150
 Integer wavelet transform, 221–222
 Internet
 efficient web page storage, 272–273
 fax standard, 402
 HTTP performance improvement, 272
 MPEG-4 transportation on, 422–423
 Intervals
 in arithmetic encoding process, 106–109
 positive-length and disjoint, 118–119
 rescaling, 114–115, *see also* Renormalization
 sequences represented by, 19–24
 Intrapixel differencing, 389
 Invariance Theorem, 40, 45
 Inverse arithmetic operations, 120
 Inversion frequencies, 188
 Inversion ranks, 190
 Irreversible color transform, JPEG2000, 354–355
 ITU-T Recommendations
 Group 3 (T.4), 398–399
 Group 4 (T.6), 399
- J**
- JBIG
 data structure and formatting, 336–338
 decoding, 338–339
 encoding
 adaptive arithmetic coding, 336
 bottom layer typical prediction, 334
 differential layer prediction, 332–334
 model templates, 334–336
 resolution reduction, 330–332
 overview of encoding/decoding, 327–330
 standard (T.82), 399
- JBIG2
 decoding control and data structures, 346–348
- generic refinement region decoding, 342–343
 generic region decoding, 341–342
 halftone region decoding, 346
 overview, 339–341
 pattern dictionary decoding, 345–346
 standard (T.88), 399
 symbol dictionary decoding, 343–345
 text region decoding, 345
- JPEG2000**
 features, 352–354
 compressed domain image processing/editing, 353
 progression, 353–354
 performance, 366–369
- JPEG2000 algorithm**
 bit-plane coding, 360–364
 JPEG2000 codestream, 365–366
 packets and layers, 364–365
 quantization, 358–360
 tiles and component transforms, 354–355
 wavelet transform, 355–358
- JPEG hardware**, 417
- JPEG-LS**
 entropy coding, 307–309
 modifications for multicomponent images, 304–306
 overview, 301–302
 prediction error correction, 306
 single-component images, 303–304
- K**
- Karp–Rabin fingerprints, 164–165
 Kolmogorov complexity
 basic definitions, 39–40
 computational issues, 44–47
 historical perspective, 51
 incompressibility, 40–42
 lower-bounding, 46–47
 notion of describing data, 38
 prefix-free encoding, 42–44
 relationship to Shannon Information Theory, 47–51
 resource-bounded, 45–46
- Kraft–McMillan inequality, 84–85
 Kraft’s inequality
 in competitive optimality, 26
 for prefix code, 14–17
- L**
- Laplacian pyramid structure, reduced, 221
- Layers, JPEG2000, 364–365
 Learning-based switch predictors, 216–217
 Least frequently used heuristic, 161
 Least recently used updating, in symbol-ranking compressor, 200–201
 Least significant bits, 259–260
 Lempel–Ziv encoder, 412–415
 processing element, 414–415
 Lempel–Ziv-77 method, 159, 232–233, 273, 407–408
 delta compressors based on, 274–275
 LZU compressor, 294–296
 Unicode compressors, 294
 Lempel–Ziv-78 method, 160, 232–233, 407–408
 Length-constrained Huffman codes, 89
 Length-index preserving transform, 236–240
 Letter index transform, 242–243
 Levenshtein codes, 58–59
 Lexicographical ordering
 FGK algorithm, 91
 over source sequence, 21
 Lifting, in hierarchical lossless image coding, 221–222
 Linear first-order unit-delay predictor, 319
 Line interleaving, 305–306
 Line Not Typical, fictitious pixel, 332–333
 Lossless audio data compression
 amplitude range and segmentation, 259–260
 basic redundancy removal, 257–258
 entropy coding, 262–263
 multiple-channel redundancy, 260
 numerical implementation and portability, 263
 prediction, 260–262
 segmentation and resynchronization, 263–264
 software systems
 MLP, 266
 PCA, 266–267
 Shorten, 265
 speed and complexity, 264–265
 variable bit rate, 264
 Lossless compression algorithms
 dictionary methods, 232–233
 performance comparison, 233–234
 statistical methods, 229–232
 transform-based methods, 233

Lossless image compression
bilevel, *see* JBIG; JBIG2
combined predictors, 217–219
error modeling, 212
hierarchical prediction, 211–212
JPEG-LS
 decoding, 309
 encoding, 302–309
 overview, 301–302
probability mass function
 prediction, 208
pyramid coding scheme, 220–222
scanning techniques, 212–214
spatial prediction, 209–211
switched predictors, 214–217
Lossy audio compression, 256
Low-entropy options, adaptive
 entropy coder, 316–317
LSI Logic products, 428–430
Lucent system-layer decoder, 428
LZW algorithm, 160

M

Macro schemes, dynamic dictionary
 methods, 157
Magnitude refinement pass, bit-plane
 coding, 363–364
MARVLE architecture, 407
 elements of, 411–412
 memory mapping, 410
Mask layer, facsimile, 401–402
Maximum ambiguity, of current and
 reference files, 284–285
Median edge detection, predictor
 based on, 216
Memory-efficient algorithms, 95
Memoryless sources, 10
 fixed-length codes for, 26–34
 variable-length codes for,
 11–18
Memory mapping, 409–410
Meridian Lossless Packing, 266
Minimum description length
 principle, 39
Mixed Raster Content (MRC)
 standard, 399–402
MMR coding, JBIG2, 340–341
Model templates, JBIG encoding,
 334–336
Modem, compression standards,
 166
Modified Huffman algorithm, 393
Modified Huffman codes, 87
Modified READ algorithm, 393–397
Most significant bits, 259
 JPEG2000, 360–361
Motorola, MPEG-2 video decoder,
 436

Move-to-Front recoding, 170,
 174–176, 185, 190
Move-to-Front step
 Burrows–Wheeler, 183–185
 elimination, 187–189
 BWT, 233
MOVIE project, 425
MPEG-2
 digital audio/video decoder by
 IBM, 436–437
 video decoder by Motorola,
 436
MPEG-4
 decoder by Sigma Designs, Inc.,
 438
 products by Toshiba, 437–438
 transportation on Internet,
 422–423
 video codec by Toshiba, 423–425
MPEG 1 Layer 3, 256
Multicomponent images,
 modifications for, 304–306
Multiple-channel redundancy, 260
Multiple-Image Network Graphics,
 388–390
Multiplications, inexact, 115–118

N

N-ary Huffman codes, 83–84
NEAR, JPEG-LS, 302, 306, 309
NEC audio video codec, 430–431
Nested loops, 389
Nested subintervals, 119–120
Noiseless source-coding theorem,
 Shannon’s, 48, 50
Noncomputability, 36–37
None filter, 380
Non-prefix codes, subtle problems
 with, 43–44
Notation, arithmetic coding,
 103–104
Number index transform, 241–243

O

Occam’s Razor, 39
Omega code, Elias, 59
One-step lookahead, greedy parsing
 with, 156–157, 164
Optimality
 arithmetic coding, 111–112
 competitive, 25–26
Optimal tree-based search,
 143–144
Order-3 Fibonacci code, 69–70
Original external pointer macro
 scheme, 157

Original pointer macro scheme, 157
Output fragments, good and bad,
 185–186

P

Packetization algorithm, MPEG-4,
 422–423
Packets, JPEG2000, 364–365
Packet telemetry, 248
Paeth filter, 380–383
Palindrome, complementary, 162
Parsing
 bidirectional, 161
 issues in dictionary construction,
 155–157
 modifications, 160–161
Patch, of minimal size, 270
Pattern dictionary, decoding,
 345–346
Peak bit rate, 264
Peer-to-peer systems, 280
Perceptual coding, lossy,
 255–256
Perfect Clarity Audio, from Sonic
 Foundry, Inc., 266–267
Performance
 HTTP, improvement, 272
 Huffman codes, 84–86
 JPEG2000, 366–369
 lossless compression algorithms,
 comparison, 233–234
 timing, degraded, 240–241
Per letter entropy, 11
Permutation, Burrows–Wheeler,
 173–174, 181–183
PNG, *see* Portable Network Graphics
Polynomial representations,
 56–57
Portable Network Graphics
 compression engine, 374–376
 compression tests and
 comparisons, 385–388
 design decisions, 372–374
 filters, 378–383
 historical background, 371–372
 MNG, 388–390
 practical compression tips,
 383–385
 zlib format, 376
 zlib library, 376–378
Prediction
 bottom layer typical, 334
 differential layer, 332–334
 lossless audio data compression,
 260–262
process, symbol-ranking
 compressors, 198–199

Prediction by Partial Match (PPM)
 Burrows–Wheeler transform and,
 180
 constant-order compressor, 295
 finite-context statistical
 compressors, 293
 method C, 231
 method D, 237–241
 Prediction error correction, 306
 Prediction error mapper, 320
 Predictors
 combined, 217–219
 FIR and IIR, 262
 linear and nonlinear, 211
 Paeth, 380–383
 preprocessor, 318–319
 Shannon coder, 196
 switched, 214–217
 symbol, 198
 using previous pixel in image, 210
 Prefix codes
 codewords in, 13–14
 in Kolmogorov complexity, 42–44
 Kraft’s inequality for, 14–16
 sharing of prefixes, 81–83
 Preprocessing
 Burrows–Wheeler compression,
 181
 prediction error mapper, 320
 predictor, 318–319
 reference file, 274
 reference sample, 319–320
 Probabilities, *see also* Error
 probability
 consistent, 10
 for memoryless sources, 27
 shift invariant, 11
 static and dynamic, 228–229
 Probability distribution
 combined, 217
 dyadic, 24–26
 given for free, 49–50
 over source sequences, 23
 Probability domain blending, of
 subpredictors, 219
 Probability mass function,
 prediction, 208
 Processing elements, Lempel–Ziv,
 413–415
 Programmability, hardware systems,
 407
 Programmable logic, 96
 Programming language, *see*
 Descriptive language
 Progressive-compatible sequential
 mode, JBIG, 328, 330
 Progressivity dimensions,
 JPEG2000, 353–354
 Pyramid coding scheme, 211–212

Q
 Quality progressive representations,
 340
 Quantization
 JPEG2000, 358–360
 and prediction error correction,
 306
R
 Recoding, MTF, 170, 174–176, 185,
 190
 Reconstruction
 and prediction error correction,
 306
 progressive, in JBIG transmission,
 328
 Recursively enumerable set, 45
 Reduced-pyramid structure, 221
 Redundancy
 coding, acceptable, 22–23
 individual and expected, 16
 multiple-channel, 260
 removal, 257–258
 Reference bitmap, refined,
 342–345
 Reference file, delta compression,
 270–271
 choosing, 278–279
 preprocessing, 274
 Reference sample, preprocessor
 predictor, 319–320
 Remote file synchronization
 applications, 279–280
 balanced pairs, 283–284
 database records and file systems:
 reconciling, 286–287
 distance measures, 283
 results, 285
 estimation of file distances, 286
 fundamental results, 284–285
 problem definition, 270–271
rsync algorithm, 280–281
 experimental results,
 282–283
 Renormalization, 121–132,
 142–143
 Rescaling, interval, 114–115, *see*
also Renormalization
 Resolution reduction, JBIG
 encoding, 330–331
 Resynchronization, segmentation
 and, 263–264
 Reverse binary tree, 408–409
 Reversible color transform,
 JPEG2000, 354–355
 Revision Control System, software
 package, 271–272
R
 Rice codes, 60–62, 75, *see also*
*e*_Rice algorithms
rsync algorithm, 280–281
 experimental results, 282–283
 Run-length coding, 179–180
 Run-length color encoding, 398
 Run-lengths, and Wheeler 1/2 code,
 73–75
 Run mode, JPEG-LS, 304
 entropy coding, 307–309
S
 Sample interleaving, 306
 Scalability, architecture, 406
 Scanning techniques, in lossless
 image compression, 212–214
 Second-extension option, adaptive
 entropy coder, 316
 Segmentation
 amplitude range and, 259–260
 resynchronization and, 263–264
 Self-delimiting codes, 55
 Self-delimiting process, 43
 Sequences
 biomolecular, compression,
 162–163
 complex nonrandom, 37
 empty, 9
 representation by intervals, 19–24
 of sources, infinite, 48–49
 Settled symbols, 131
 Shannon coder, 196
 Shannon–Fano coding, 80
 Shannon Information Theory
 complex nonrandom sequences, 37
 relationship to Kolmogorov
 complexity, 47–51
 structured random strings, 37–38
Shorten software, 265
 Sigma Designs, Inc., MPEG-4
 decoder, 438
 Single-chip decoder, by Zoran, 435
 Single-component images, 303–304
 Single-pass methods, 158
 Single-progression sequential mode,
 JBIG, 328
 Software programs, lossless audio
 data compression, 265–267
 Sorting algorithm, improvements to,
 182–183
 Source coding system
 fixed-length codes for memoryless
 sources, 26–34
 variable-length codes for source
 symbols, 12–13
 Source modeling, separation from
 coding, 113–114
 Source samples, blocks, 312–313

- Source symbols
 blocks, 17–18
 variable-length codes for, 12–13
- Space applications
 adaptive entropy coder, 313–317
 coded data format, 321
 e.Rice algorithm, 312–313
 implementation issues and
 applications, 324–326
 lossless decoder, 321–323
 preprocessing stage, 318–320
- Space-constrained delta
 compression, 277–278
- Spatial prediction, in lossless image compression, 209–211
- Speed-efficient algorithms, 95–96
- Splay tree algorithm, 93–94
- Split-sample option, adaptive entropy coder, 314–315
- Star (*) transformation, 234–236
- Start-step-stop codes, 64–65
- Static dictionary, 154–155
- Stationary sources, discrete, 10–11
- Statistical coding
 Burrows–Wheeler compressor, 176–180
 for lossless algorithms, 229–232
- Statistical compressors, 185–187
 finite- and unbounded-context, 293
- Sticky MTF, 184–185
- STMicroelectronics, video decoder, 433–435
- String-matching, PNG, 378
- Strings
 algorithmic prefix complexity, 42–43
 incompressible, 41–42
 run-length compressed, 73–75
 searching, fingerprints for, 164–165
 vs. sources, 37
 structured random, 37–38
- Stripes, document partitioned into, 400–401
- Structured coding model, 179
- Sub filter, 380
- Subpredictors, combining, 217–219
- Suffix-complete dictionary, 156
- Suffix trees
 as alternative to explicit sort, 182–183
 construction, 163–164
- Summation codes, 71–73
- Switched predictors, 214–217
 explicit detection-based, 215–216
 learning-based, 216–217
- Symbol dictionary, decoding, 343–345
- Symbol-ranking compression, 195–201
 and Burrows–Wheeler compression, 200
 compressor history, 197
 fast, 200–201
 prediction flagging, 197–198
 Shannon coder, 196
- Symbol search
 bisection search, 143
 on sorted symbols, 144–145
 optimal tree-based, 143–144
 sequential search on sorted symbols, 143
- Synchronization, *see also* Remote file synchronization
 encoder–decoder decisions, 113
 file, BWT in, 189
- T
- Tandem repeats, 162
- TANGRAM coprocessor, 420
- Telemetry, description of, 247–250
- Telemetry compression
 existing, 252–253
 issues in, 250–251
- Ternary comma codes, 70–71
- Ternary polynomial representation, 56–57
- Test compressors, 294
- Text compression
 ACB algorithm of Buynovsky, 201–203
 classification of lossless compression algorithms, 229–234
 hardware, 407–415
 Lempel–Ziv encoder, 412–415
 tree-based encoder, 408–412
 initial letter preserving transform, 241–243
 length-index preserving transform, 236–240
 letter index transform, 241–243
 number index transform, 241–243
 star (*) transformation, 234–236
 timing performance
 measurements, 240–241
- Text region, decoding, 345
- Textual substitution, compression with, 153
- TIFF-FAX standard, 402
- Tile-parts, 366
- Tiles, JPEG2000, 354, 364
- Timing performance, degraded, 240–241
- Toshiba
 MPEG-4 products, 437–438
 MPEG-4 video codec, 423–425
- Transform-based methods, for lossless algorithms, 233
- Tree-based encoder, 408–412
 MARVEL architecture, 411–412
 memory mapping, 409–410
- Tree-structured codebook, Huffman, 81–83
- Trie–reverse trie pairs, 164
- Tries, and compact tries, 163
- Tunstall code, 88–89
- U
- Unary codes, 57–58, 307
- Uncertainty, resolved through information, 5–6
- Unicode
 character codings, 291–293
 big-endian vs. little-endian, 292
 UTF-8 coding, 292–293
- compression
 comparison of compressors, 295–296
 finite- and unbounded-context statistical compressors, 293
 LZ77 compressors, 294
 file test suite, 294–295
- Unidirectional dictionary methods
 construction modifications, 160
 dictionary maintenance in dynamic methods, 161
 historical overview, 158
 Lempel–Ziv-77, 159
 Lempel–Ziv-78, 160
 LZ77 variants, 159
 parsing modifications, 160–161
- Uniquely decipherable code, 230
- Uniquely decodable code, 13–14
- Universal codes
 characteristics, 55–56
 comparison of representations, 75–77
 Elias omega and Even–Rodeh codes, 59–60
 Fibonacci codes, 65–70
 Golomb codes, 62–64
 Levenshtein and Elias gamma codes, 58–59
 polynomial representations, 56–57

- Rice codes, 60–62
start-step-stop codes, 64–65
summation codes, 71–73
ternary comma codes, 70–71
unary codes, 57–58
Wheeler 1/2 code and run-lengths, 73–75
Up filter, 380
User files, synchronization, 279–280
- V**
- Variable-length codes
 for memoryless sources, 11–18
 for sources with memory, 18–26
Varn's algorithm, 97
vcdiff, compared to *zdelta* and *xdelta*, 275–277
Video compression hardware, 417–442
 commercial video and audio products, 426–442
- W**
- Wavelet architectures, 416–417
Wavelet transform, JPEG2000, 355–358
Web page
 commercial, for vendors, 442
 efficient storage, 272–273
Wheeler 1/2 code, 74–75
- X**
- xdelta*, compared to *zdelta* and *vcdiff*, 275–277
Xdelta File System, 272
- Z**
- zdelta*, compared to *xdelta* and *vcdiff*, 275–277
Zeckendorf representation, 66
Zero-block option
 adaptive entropy coder, 316–317
 coded data format, 323
 testing and, 324
Zero-frequency leaf, 93
Zip variants, of LZ77, 159
Ziv–Lempel dictionary-based scheme, 89, 97, 232–233, *see also* Lempel–Ziv–77 method
zlib, format and library, 376–378
Zoran, single-chip decoder, 435

This Page Intentionally Left Blank

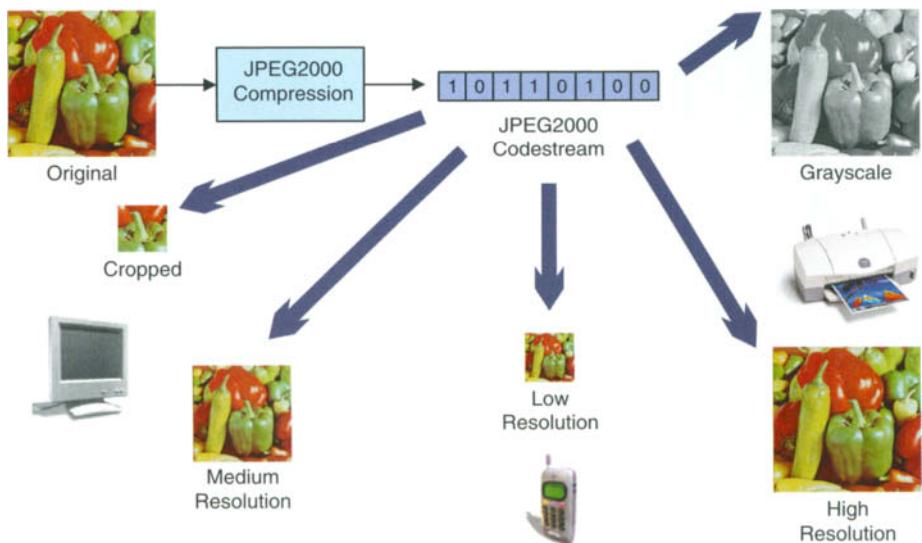


FIGURE 18.1

Some of the functionality offered by JPEG2000.



FIGURE 18.14

Peppers image compressed at very low bit-rates using JPEG and JPEG2000. (a) JPEG compressed at 0.088 bpp. (b) JPEG compressed at 0.155 bpp. (c) JPEG 2000 compressed at 0.088 bpp. (d) JPEG 2000 compressed at 0.155 bpp.

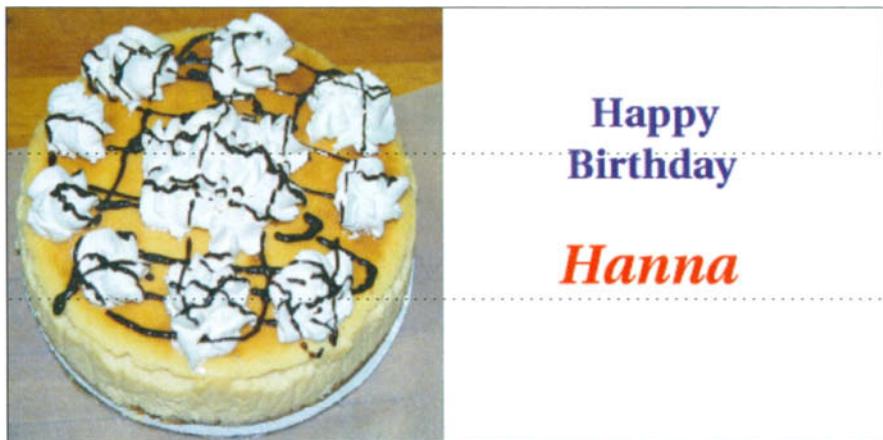


FIGURE 20.5
Test image.



FIGURE 20.6
Background for the test image.



FIGURE 20.7
Foreground for the test image.

This Page Intentionally Left Blank