FAST PARALLEL ALGORITHMS FOR UNIVERSAL LOSSLESS SOURCE CODING

Dror Baron, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2003
Prof. Yoram Bresler, Adviser

Most data compression research in recent years has focused on lossy compression for audio, images, and video, but lossless source coding is still important for compressing text files, executables, financial and medical data, etc. When the statistics of the source are unknown, a universal method that estimates a model for the source must be used. This dissertation focuses on fast algorithms for universal lossless source coding.

We first identify inherent redundancies in previous uses of the Burrows Wheeler transform (BWT), an invertible permutation transform that has been suggested for lossless compression, and offer several improvements to the previous state of the art in BWT-based compression and semipredictive encoding. These improvements yield an $O(N)$ nonsequential semipredictive encoder whose redundancy with respect to any (unbounded depth) tree source is $O(1)$ bits per state above Rissanen’s redundancy bound.

We then develop parallel algorithms for universal lossless coding. We first bound the redundancy of two-part codes for independent and identically distributed sequences, and show how two-part codes can be used for distributed compression. We then describe our parallel compression algorithm, which is the main contribution of the dissertation. We partition the length-$N$ input into $B$ blocks, accumulate statistical information on all $B$ blocks in parallel, estimate the single minimum description length (MDL) source underlying all $B$ blocks, and encode the blocks in parallel. We provide an $O(N/B)$ complexity parallel algorithm that compresses almost as well as the best serial algorithms.
Our last contribution is a new suffix lists data structure that leads to several efficient algorithms for implementing the BWT. The distinguishing feature of our algorithms is that they are simple enough to be implemented in hardware.

The $O(N/B)$ parallel compression algorithm estimates the MDL source among all tree sources whose maximal depth is $\log(N/B)$. This algorithm can be extended to parallel algorithms that support unbounded context depths. This will provide low redundancy performance over a much broader class of sources, and may lead to new applications that until now were limited by the throughput bottleneck of serial compression algorithms.
FAST PARALLEL ALGORITHMS FOR UNIVERSAL LOSSLESS SOURCE CODING

BY

DROR BARON

B.Sc., Technion - Israel Institute of Technology, 1997
M.Sc., Technion - Israel Institute of Technology, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2003

Urbana, Illinois
ABSTRACT

Most data compression research in recent years has focused on lossy compression for audio, images, and video, but lossless source coding is still important for compressing text files, executables, financial and medical data, etc. When the statistics of the source are unknown, a universal method that estimates a model for the source must be used. This dissertation focuses on fast algorithms for universal lossless source coding.

We first identify inherent redundancies in previous uses of the Burrows Wheeler transform (BWT), an invertible permutation transform that has been suggested for lossless compression, and offer several improvements to the previous state of the art in BWT-based compression and semipredictive encoding. These improvements yield an $O(N)$ nonsequential semipredictive encoder whose redundancy with respect to any (unbounded depth) tree source is $O(1)$ bits per state above Rissanen’s redundancy bound.

We then develop parallel algorithms for universal lossless coding. We first bound the redundancy of two-part codes for independent and identically distributed sequences, and show how two-part codes can be used for distributed compression. We then describe our parallel compression algorithm, which is the main contribution of the dissertation. We partition the length-$N$ input into $B$ blocks, accumulate statistical information on all $B$ blocks in parallel, estimate the single minimum description length (MDL) source underlying all $B$ blocks, and encode the blocks in parallel. We provide an $O(N/B)$ complexity parallel algorithm that compresses almost as well as the best serial algorithms.
Our last contribution is a new suffix lists data structure that leads to several efficient algorithms for implementing the BWT. The distinguishing feature of our algorithms is that they are simple enough to be implemented in hardware.

The $O(N/B)$ parallel compression algorithm estimates the MDL source among all tree sources whose maximal depth is $\log(N/B)$. This algorithm can be extended to parallel algorithms that support unbounded context depths. This will provide low redundancy performance over a much broader class of sources, and may lead to new applications that until now were limited by the throughput bottleneck of serial compression algorithms.
ACKNOWLEDGMENTS

I want to thank everybody who has contributed towards this work. First I want to thank my adviser, Yoram Bresler, for his great efforts in guiding me, proofreading our papers, and providing assistance when necessary. I also want to thank the other committee members, Richard Blahut, Bruce Hajek, Sean Meyn, and Andrew Singer for reading and commenting on the manuscript. In particular, thanks to Andy for lots of advice and for helping me organize and chair sessions at the Allerton conference. Thanks also to the National Science Foundation, which funded most of my Ph.D. work through grant nos. MIP-97-07633, CDA-96-24396, and CCR-01-22293, and to the M. E. Van Valkenburg Graduate Research Award, which supported me during 2002.

I am grateful to additional faculty at the University of Illinois for various forms of assistance. Thanks to Christoforos Hadjicostis, Ralf Koetter, Doug Jones, Pierre Moulin, Dave Munson, Naresh Shanbhag, and Pramod Viswanath. In particular, Pierre taught several classes that I attended, helped with the painful faculty applications process, and invited me to his research group’s weekly meetings.

My work at the University of Illinois was also aided by the support staff here, including Jeri Borchers, Vicki Brown, Sheryle Carpenter, Beth Dennison, Terri Hovde, Gabriel Lopez-Walle, Mary Parsons, Jeanne Reinhart, Rhonda Stebbins, Dave Washburn, and Deanna Zachary. I am sure that Deanna will not have as many rooms to reserve for talks, nor will she need to address as many questions about account numbers, after I leave.
I also thank faculty at the Technion - Israel Institute of Technology, who got me to the point where I could come to Illinois and start making research contributions almost immediately. Thanks to Yitzhak Birk for supervising my M.Sc. work and helping at various stages of my career. Thanks also to David Malah, Neri Merhav, Nahum Shimkin, and Ofer Zeitouni for their assistance.

I am also thankful to additional researchers that I was in contact with over the years. Thanks to Bernhard Balkenhol at the University of Bielefeld, Toby Berger at Cornell, Ioannis Kontoyiannis at Brown University, Stefan Kurtz at the University of Bielefeld, N. Jesper Larsson at Apptus Technologies, Ragnar Nohre at Jönköping University, Kannan Ramchandran at Berkeley, Gil Shamir at the University of Utah, Dafna Sheinwald at IBM, Sergio Verdú at Princeton, Marcelo Weinberger at HP, and Frans Willems at Eindhoven. Thanks also to people at Witcom, Ltd., where I spent two years and had many beneficial experiences.

I want to thank my officemates, co-students, and groupmates at the University of Illinois and at the Technion for their support. Thanks to Nitin Aggarwal, Samit Basu, Jeff Brokish, Laurent Couvreur, Ashvin George, Armin Gerstberger, Yael Hochberg, Prakash Ishwar, Aleksandar Ivanovic, Soumya Jana, Negar Kiyavash, Suleyman Kozat, Aaron Lanterman, Tie Liu, M. Kivanc Mihcak, Rob Morrison, Jill Nelson, Subramanya G. Prasad, Ananya Sen Gupta, Sanjay Shakkottai, Nachum Shamir, Ilan Sutzkover, Raman Venkataramani, Tsachy Weissman, Jong Chul Ye, and Serdar Yuksel. Thanks also to many friends who helped during the years. For fear of forgetting several, I prefer to skip the list!

Finally, thanks to my family for their continued support. My Ph.D. work was motivated by my mother who claimed that I am incapable of living alone for several years and completing the degree. Not to be overlooked is my dear cousin Opher Baron who has been a great help in numerous ways.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td></td>
</tr>
<tr>
<td>1.3.1</td>
<td></td>
</tr>
<tr>
<td>1.3.2</td>
<td></td>
</tr>
<tr>
<td>1.3.3</td>
<td></td>
</tr>
<tr>
<td>1.3.4</td>
<td></td>
</tr>
<tr>
<td>1.3.5</td>
<td></td>
</tr>
<tr>
<td>1.3.6</td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>1.4.1</td>
<td></td>
</tr>
<tr>
<td>1.4.2</td>
<td></td>
</tr>
<tr>
<td>1.4.3</td>
<td></td>
</tr>
<tr>
<td>1.4.4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>2.2.1</td>
<td></td>
</tr>
<tr>
<td>2.2.2</td>
<td></td>
</tr>
<tr>
<td>2.2.3</td>
<td></td>
</tr>
<tr>
<td>2.2.4</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>2.3.1</td>
<td></td>
</tr>
<tr>
<td>2.3.2</td>
<td></td>
</tr>
<tr>
<td>2.3.2.1</td>
<td></td>
</tr>
<tr>
<td>2.3.2.2</td>
<td></td>
</tr>
<tr>
<td>2.3.2.3</td>
<td></td>
</tr>
<tr>
<td>2.3.3</td>
<td></td>
</tr>
<tr>
<td>2.3.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>44</td>
</tr>
</tbody>
</table>
2.4 Properties ......................................................... 45
  2.4.1 Redundancy ............................................... 45
  2.4.2 Optimality and statistical inference ...................... 46
  2.4.3 Comparison to other source coding methods .............. 47
  2.4.4 Decoder memory versus compression tradeoff .............. 47
2.5 Conclusions and Future Work .................................. 49

3 TWO-PART CODES WITH LOW WORST-CASE REDUNDANCIES FOR
  DISTRIBUTED COMPRESSION OF I.I.D. SEQUENCES ............... 51
  3.1 Introduction .................................................. 51
  3.2 Bounds on Universal Coding of Bernoulli
    Sequences .................................................... 53
  3.3 Quantized-Parameter Two-Part Codes ......................... 56
    3.3.1 Structure of quantized-parameter (QP) two-part codes .... 56
    3.3.2 Using quantized-parameter codes for distributed source coding ... 57
    3.3.3 Divergence and redundancy of quantized-parameter codes .... 59
  3.4 Code Motivated by Jeffreys’ Prior .......................... 62
    3.4.1 Quantization bins and representation levels .......... 65
    3.4.2 Worst-case divergence .................................. 65
    3.4.3 Number of bins .......................................... 66
    3.4.4 Operational Jeffreys code ............................... 67
  3.5 Optimal Two-Part Codes ...................................... 68
    3.5.1 Necessary conditions for optimality .................... 68
    3.5.2 An attainable lower bound on the worst-case redundancy .... 71
    3.5.3 Modified Jeffreys method ................................ 77
  3.6 Finite Alphabets ............................................. 78
    3.6.1 Rissanen’s bound for finite alphabets .................. 79
    3.6.2 Two-part codes for finite alphabets .................... 80
    3.6.3 Two-part codes with priors ............................. 84
  3.7 Discussion .................................................. 89

4 A WORK-EFFICIENT PARALLEL TWO-PASS MDL CONTEXT TREE
  ALGORITHM FOR UNIVERSAL SOURCE CODING ..................... 91
  4.1 Motivation and Background ................................... 91
    4.1.1 Motivation ............................................... 91
    4.1.2 Design space ............................................ 92
    4.1.3 Previous work on hardware compression and parallel
          compression .............................................. 93
      4.1.3.1 Hardware compression ................................ 94
      4.1.3.2 Parallel compression ............................... 95
    4.1.4 Naive parallelization ................................... 96
    4.1.5 Parallelizing the two-pass MDL approach ................. 97
  4.2 Basic Algorithm ............................................. 98
4.2.1 Computational model ........................................... 98
4.2.2 General structure ............................................. 99
4.2.3 Parallel source description .................................. 101
   4.2.3.1 Two-part codes in the PTP-MDL algorithm .......... 101
   4.2.3.2 Worst-case redundancy of two-part codes .......... 103
   4.2.3.3 Number of quantization bins ....................... 104
   4.2.3.4 Source description in Phase I and Phase II ........ 105
   4.2.3.5 MDL tree source .................................... 107
4.2.4 Phase I ....................................................... 107
   4.2.4.1 Constructing context trees (computing symbol counts) . 107
   4.2.4.2 Computing the MDL length .......................... 109
   4.2.4.3 Computing the MDL length with finite precision .... 111
4.2.5 Phase II ...................................................... 112
4.2.6 Encoder output stage ...................................... 113
4.2.7 Decoder ....................................................... 115
4.3 An $O(N/B)$ Complexity Algorithm .......................... 116
   4.3.1 Determining the generators of all the symbols ....... 116
   4.3.2 Adding up block symbol counts ....................... 118
   4.3.3 Pipelining ................................................. 119
4.4 Properties ....................................................... 120
   4.4.1 Computational complexity and memory use .......... 120
   4.4.2 Redundancy analysis ................................... 122
   4.4.3 Tradeoffs between performance measures ........... 124
   4.4.4 Random access compression ............................ 125
4.5 Extensions ....................................................... 126
   4.5.1 Advanced parallel source description ................ 126
   4.5.2 Practical aspects .................................... 128
   4.5.3 Overlapping blocks ................................... 128
4.6 Discussion ..................................................... 129

5 LOW COMPLEXITY SEQUENTIAL SUFFIX SORTING FOR
BWT-BASED DATA COMPRESSION ...................................... 132
5.1 Introduction .................................................... 132
   5.1.1 Suffix sorting and compression ........................ 132
   5.1.2 Lossless compression in VLSI .......................... 133
   5.1.3 Overview of chapter ................................... 134
5.2 Suffix Sorting and the BWT .................................. 136
   5.2.1 Suffix sorting .......................................... 136
   5.2.2 State of the art ....................................... 136
   5.2.3 A suffix sorting perspective on the BWT .......... 138
5.3 Basic Algorithm ............................................... 138
   5.3.1 Suffix lists ............................................ 139
   5.3.2 The core of the algorithm ............................ 139
APPENDIX H PROOF OF THEOREM 11 .......................... 201
APPENDIX I PROOF OF THEOREM 12 .......................... 203
APPENDIX J PROOF OF LEMMA 7 .............................. 205
APPENDIX K EXAMPLE OF THE PMDL ROUTINE .............. 207
APPENDIX L PROOF OF LEMMA 8 .............................. 210
REFERENCES ..................................................... 212
VITA ................................................................. 217
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>BWT-MDL encoder vs. other source coding algorithms.</td>
</tr>
<tr>
<td>5.1</td>
<td>The memory use of hierarchical lists (in bytes) on current typical computer architectures. (The basic and bidirectional algorithms both use 9 bytes per suffix.)</td>
</tr>
<tr>
<td>5.2</td>
<td>Files used for simulation. (File sizes are in bytes.)</td>
</tr>
<tr>
<td>5.3</td>
<td>Total aggregate accesses to the list, and aggregate accesses for Case 3, performed by the basic, bidirectional, and hierarchical algorithms, for different alphabet sizes. (Run times are in seconds.)</td>
</tr>
<tr>
<td>5.4</td>
<td>Run times for the basic (B), bidirectional (BD), and hierarchical (H) algorithms; suffix trees by Kurtz (K); doubling method by Larsson-Sadakane (LS); ternary-split by Seward (S). (Run times are in seconds.)</td>
</tr>
</tbody>
</table>


**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Caption</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A tree source.</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>The prefix tree for $x_0^{11} = 01011111111$.</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>A portion of an atomic prefix tree (right) may have more nodes than the</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>corresponding portion of a compact prefix tree (left).</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Pseudocode for the Basic BWT-MDL encoder. The bounded depth $MDL$</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>routine is described in Figure 2.2.</td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Pseudocode for the bounded depth $MDL$ routine. We use a binary alphabet</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>to keep things simple.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Pseudocode portion for the unbounded variant of the $MDL$ routine.</td>
<td>34</td>
</tr>
<tr>
<td>2.4</td>
<td>The relation between segments of $y$ and nodes of the (compact) prefix</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>tree $T$.</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>The structure of $t'$, $\hat{t}$, $t$, and children nodes of $t$</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>shows that keeping the children states of $t$ requires keeping the arc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>from $t'$ to $t$, while pruning the tree requires keeping the length-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arc from $t'$ to $\hat{t}$. The illustration uses an atomic structure</td>
<td></td>
</tr>
<tr>
<td></td>
<td>to show the location of “invisible” nodes.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>A geometric interpretation of bin divergences.</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>The $\overline{WCR}$ above Rissanen’s bound of quantizer-optimal two-part</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>codes approaches $\frac{1}{2} \log \left( \frac{e^3}{2} \right) = 1.047$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bits.</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>WCBDs for $K = \sqrt{N}$. Away from the boundaries of the parameter</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td>space $\Omega$, $D(b_{a-1}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>are large WCBDs near the boundaries of $\Omega$, but these regions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>become narrower as $N$ increases.</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>The constructed quantizer $\hat{J}_K$ and the Jeffreys quantizer $J_K$.</td>
<td>75</td>
</tr>
<tr>
<td>3.5</td>
<td>The $\overline{WCR}$ above Rissanen’s bound of the Jeffreys code</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td>approaches 1.221 bits. The $\overline{WCR}$ of quantizer-optimal codes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and modified Jeffreys codes, and the $WCR$ of operational Jeffreys</td>
<td></td>
</tr>
<tr>
<td></td>
<td>codes, approach 1.047 bits above Rissanen’s bound.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Block diagram of the PTP-MDL encoder.</td>
<td>99</td>
</tr>
<tr>
<td>4.2</td>
<td>Block diagram of Phase II.</td>
<td>100</td>
</tr>
<tr>
<td>4.3</td>
<td>Block diagram of a decoding block.</td>
<td>101</td>
</tr>
</tbody>
</table>
4.4 Pseudocode for computing the block symbol counts of all leaf contexts. . . 109
4.5 Pseudocode for the PMDL routine. ........................................ 111
4.6 Block diagram of an interleaving scheme in the encoder. ............... 114
4.7 Block diagram of the decoder. ............................................ 115
4.8 Pseudocode for constructing the generator table \( g \), where \( g_j = s \) if \( s \) is a suffix of the length-\( D_{\text{max}} \) context whose context index is \( j \). ................. 118
4.9 Pseudocode for Phase II. The decoder is similar. ....................... 119
4.10 An adder tree in VLSI adds up \( B \) terms in \( O(\log(B)) \) time. With pipelining, a symbol count is generated every \( O(1) \) clock cycles. ................. 120
4.11 An outline of the \( O(N/B) \) complexity PTP-MDL encoder. .......... 121

5.1 A typical BWT-based compression system. ................................ 133
5.2 Structure of a doubly linked suffix list. The basic algorithm scans upwards, examining increasing suffixes, up to the head. ................. 140
5.3 Pseudocode for the basic algorithm. Global variables are head, tail, and temp pointers, the \( i \) index, and \( c \) and \( o \), which are symbols. ....... 144
5.4 A Markov source that at each step either transmits the symbol 0 or the two symbols 10, each with probability 0.5. (We label state transitions with the transition probability followed by the output). ................. 161
5.5 The structure of hierarchical lists. (Pointers inside the same list are solid lines, and pointers in between hierarchies are dashed.) ............ 164
5.6 Block diagram of a proposed VLSI implementation of the bidirectional algorithm. ......................................................... 168
5.7 The effect of the block length \( K \) on the performance of the hierarchical algorithm. (Run times are in seconds.) ......................... 174
CHAPTER 1

INTRODUCTION AND BACKGROUND

1.1 Introduction and Overview

In spite of the focus in recent years on lossy compression for audio, images, and video, lossless data compression remains crucial in applications such as text files, facsimiles, software executables, medical and financial data, etc. Further applications for lossless compression, and a collection of related papers, appear in the November 2000 Special Issue of the Proceedings of the IEEE.

Universal source coding algorithms, which deal with sources whose statistics are unknown, are of particular importance. When the probability model for the finite alphabet source generating the data is given, Shannon codes and arithmetic coding [1] allow one to asymptotically approach the entropy rate for the source. However, in most cases we do not know the statistics of the source. For example, although the research on text compression is vast, no single model can be used, because different people use different vocabularies and have different writing styles. Universal coding methods are designed for universal performance over a broad class of possible sources. In these methods the source parameters are estimated, either implicitly or explicitly, and the sequence itself is encoded accordingly. Because part of the encoding describes the parameters, the performance of universal codes is worse than that of Shannon codes that have been optimized.
for the specific source. Rissanen [2] proved that for a source with \( \nu \) (unknown) parameters over a compact set, for length-\( N \) input sequences, the expected redundancy \( \rho \) over the entropy rate is at least \( \frac{\nu}{2} \log_2(N) \) bits, except for a set of parameters whose volume vanishes as \( N \to \infty \).

Much of the research effort in universal data compression has gone into reducing the redundancy \( \rho \). Several approaches to source coding [3–14] have essentially achieved Rissanen’s bound, i.e., for tree sources with \( |S| \) states over a size-\( |\mathcal{X}| \) alphabet, \( \rho \leq |S| \left[ \frac{|\mathcal{X}| - 1}{2} \log_2(N) + O(1) \right] \) (the redundancy above Rissanen is \( O(1) \) bits per state). This theoretical advantage is accompanied by demonstrated compression performance over a range of test corpora [15].

With the quest for minimum redundancy essentially over, other aspects gain in both theoretical and practical importance. In addition to low redundancies, a universal coding method must be computationally fast, and use little memory. Also, in some applications delay is important, and nonsequential methods are undesirable. There is an inherent tradeoff between these goals; for example, memory use can be improved by compressing short blocks, but owing to the redundancies in each of these blocks, running the compression method on longer blocks (or on the entire file) often improves compression quality. As we discuss later, many existing source coding algorithms have relatively high computational complexities [5–14]. Therefore, the main goal of current research is to develop algorithms featuring fast computation and low memory use, while providing compression near the redundancy bounds.

As the convergence of practical algorithms to the theoretical bounds continues, we expect that hardware implementations will gradually gain importance. In particular, a hardware system that would provide real time compression at high data rates, without sacrificing the compression quality of software compression applications, would make lossless software compression irrelevant, except for special applications.
In this dissertation, we present several methods for universal lossless source coding. Each of the main chapters is adapted from a paper that we have submitted for publication [16–20], and is self-contained given the background material in Chapter 1. In Chapter 2, we use the Burrows Wheeler transform (BWT) [21], an invertible permutation transform that has been suggested for lossless compression [3, 4, 21–26], as an efficient computational tool for pruning context trees. We offer several improvements to the previous state of the art in BWT-based compression and semipredictive encoding. These improvements yield an $O(N)$ nonsequential semipredictive encoder whose redundancy with respect to (w.r.t.) any (unbounded depth) tree source is $O(1)$ bits per state above Rissanen’s redundancy bound. We call this algorithm BWT-MDL because it utilizes efficient BWT construction algorithms to estimate the tree source that leads to the minimal description length (MDL) of the input sequence. BWT-MDL uses this MDL source to encode the input sequence.

The first step towards parallel algorithms appears in Chapter 3, where we study two-part codes for independent and identically distributed (i.i.d.) sequences. We bound the redundancy of two-part codes, and show how they can be used for distributed compression. Chapter 4 presents our parallel compression algorithm, which is the main contribution of the dissertation. The parallel algorithm leverages our work on BWT-MDL and two-part codes. We partition the length-$N$ input sequence into $B$ blocks, accumulate statistical information on all $B$ blocks in parallel, estimate the single MDL source underlying all $B$ blocks, and encode the blocks in parallel. We provide an $O(N/B)$ complexity parallel algorithm that compresses almost as well as the best serial algorithms.

Chapter 5 describes suffix lists, a new data structure that leads to simple, fast, and memory-efficient algorithms for suffix sorting and computation of the BWT. The distinguishing feature of suffix list algorithms is that they are simple enough to be implemented in hardware. The BWT, which underlies some of the most efficient high-performance universal compression algorithms to-date [21, 24, 25], as well as our BWT-MDL algorithm,
is often their computational bottleneck. Hardware implementations of the BWT open up a new range of potential applications for fast high-performance compression.

Chapter 6 discusses our contributions and describes topics for future research. The main future direction is a parallel compression algorithm that supports unbounded context depths. This will provide low redundancy performance over a much broader class of sources, and may lead to new applications that until now were limited by the throughput bottleneck of serial compression algorithms.

The remainder of this chapter is organized as follows. Section 1.2 reviews the bounds on the performance of universal coding more formally. Section 1.3 presents the background material in source modeling and algorithmic tools that is necessary for understanding this dissertation. Section 1.4 presents related source coding methods and explains the limitations of compressing the BWT output directly.

1.2 Bounds on Universal Coding

The redundancy (or excess coding length) above the entropy rate is a measure of the compression quality of universal codes. Accordingly, lower bounds on the redundancy serve as a benchmark for compression quality. For length-$N$ sequences $x$ generated by a stationary ergodic source whose per-symbol entropy for length-$N$ blocks is $H_N$, the expected redundancy $\rho$ is defined as

$$\rho \triangleq E_x[\ell(x)] - NH_N,$$

where $\ell(x)$ is the length of a uniquely decodable code $[1]$ for $x$, and $E_x[\cdot]$ denotes expectation over all length-$N$ inputs. Rissanen $[2]$ proved that for a source with $\nu$ (unknown) parameters over a compact set, a lower bound on the expected redundancy is

$$\rho \geq \frac{\nu}{2}(1 - \epsilon)\log(N) \quad (1.1)$$
bits (\(\log(\cdot)\) denotes the base-two logarithm), for any \(\epsilon > 0\), except for a set of parameters whose volume vanishes as \(N \to \infty\). Similarly, for an individual sequence \(x\), the \textit{pointwise redundancy} \(\rho(x)\) of a code with length function \(l(x)\) w.r.t. a class \(\mathcal{C}\) of source models is defined as

\[
\rho(x) \triangleq l(x) - N\hat{H}_x,
\]

where \(\hat{H}_x\) is the \textit{maximum likelihood (ML) entropy} of \(x\) w.r.t. \(\mathcal{C}\), i.e., the entropy of \(x\) w.r.t. the best model in \(\mathcal{C}\) with parameters set to their ML estimates. Weinberger et al. [27] sharpened Rissanen’s result from a probabilistic setup to individual sequences and proved that any sequential uniquely decodable code satisfies

\[
\rho(x) \geq \frac{\nu}{2}(1 - \epsilon) \log(N),
\]

for any \(\epsilon > 0\), except for a set of inputs whose probability vanishes as \(N \to \infty\). Furthermore, they proved the existence of sequential uniquely decodable codes that attain this lower bound. For length-\(N\) sequences over a size-\(|\mathcal{X}|\) finite alphabet \(\mathcal{X}\), the \textit{worst-case redundancy} (WCR) over all possible inputs is

\[
\max_{x \in \mathcal{X}^N} \rho(x) = \max_{x \in \mathcal{X}^N} \{l(x) - N\hat{H}(x)\}.
\]

Rissanen [28] proved that, for universal coding of i.i.d. sequences, the WCR is at least \(\frac{|\mathcal{X}|-1}{2} \log(N) + C_{|\mathcal{X}|} + o(1)\) bits, where \(C_{|\mathcal{X}|}\) was specified (see Section 3.2 for details). Because i.i.d. models are too simplistic for modeling “real-life” inputs, we use tree sources instead.

### 1.3 Source Modeling and Algorithmic Tools

#### 1.3.1 Tree sources

We denote by \(|\cdot|\) either the length of a sequence or the cardinality of a set. Let \(x_i^j\) denote the sequence \(x_i, x_{i+1}, \ldots, x_j\) where \(x_k \in \mathcal{X}\) for \(i \leq k \leq j\). Consider length-\(N\) input
sequences $x = x_1^N$, i.e., $x \in \mathcal{X}^N$. Let $\mathcal{X}^*$ denote the set of finite-length sequences over $\mathcal{X}$. Define a tree source \cite{10,11,13,23,25} as a finite set of sequences called states $\mathcal{S} \subset \mathcal{X}^*$ that is complete and proper (completeness implies that any semi-infinite sequence has a suffix in $\mathcal{S}$; properness implies that there are no two sequences $s_1, s_2 \in \mathcal{S}$ such that $s_1$ is a suffix of $s_2$ \cite[13, p.654}], and a set of conditional probabilities $p(\alpha|s)$ for each state $s \in \mathcal{S}$ and each symbol $\alpha \in \mathcal{X}$. We denote a tree source by $\{\mathcal{S}, \Theta\}$, where $\mathcal{S}$ is the source structure and $\Theta = \{p(\alpha|s) : \alpha \in \mathcal{X}, s \in \mathcal{S}\}$ are the conditional probabilities. We say that $s$ generates symbols following it. Because $\mathcal{S}$ is complete and proper, the sequences of $\mathcal{S}$ can be arranged as leaves on an $|\mathcal{X}|$-ary tree \cite{29} (Figure 1.1); the unique state $s$ that generated $x_i$ can be determined by entering the tree at the root, first choosing branch $x_{i-1}$, then branch $x_{i-2}$, and so on, until some leaf $s$ is encountered. Let $D \triangleq \max_{s \in \mathcal{S}} |s|$. Then the string $x_{i-D}^i$ uniquely determines the current state $s$; the previous symbols $x_{i-L}^{i-1}$ ($L \leq D$) that uniquely determine the current state $s$ are called the context, and $L$ is called the context depth for state $s$. The quantity $D$ is the maximum context depth.

Tree sources are commonly used for modeling text \cite{15}, primarily because the last several characters often greatly limit the next character, hence its conditional entropy is small. More generally, owing to the flexibility in their structure and transition probabilities, they provide a rich class of models for finite-alphabet stationary sources in which one symbol exhibits dependencies on previous symbols. Thus, they may be good models for sequences as diverse as text \cite{15}, image pixels in an appropriately chosen scan order \cite{5,6,30}, or DNA sequences. In the remainder of the dissertation we assume that $x$ is generated by an unknown tree source $\mathcal{S}$.

**Example 1** A simple tree source defined over $\mathcal{X} = \{0, 1\}$ appears in Figure 1.1. When determining the state that generated $x_i$, we first examine $x_{i-1}$. If $x_{i-1} = 0$ we are in state 0. Else $x_{i-1} = 1$, and we also need to examine $x_{i-2}$. In this case, if $x_{i-2} = 0$ we are in
state 01, else }x_{i-2} = 1 and we are in state 11. Because 2 symbols from the past suffice to uniquely determine the state, D = 2. The states 0, 01, and 11 are associated with the conditional probabilities }p(x_i = 1|0) = 0.8, }p(x_i = 1|01) = 0.4, and }p(x_i = 1|11) = 0.9. For }N = 11, the input sequence might be }x_1^{11} = 01011111111.

1.3.2 Prefix and suffix sorting

We append a unique sentinel symbol $, larger lexicographically than any symbol in }X, at the beginning of the sequence, and define }x_0 \triangleq $, i.e., }x_0^N = $x_1^N$, where $\alpha \beta$ denotes concatenation of $\alpha$ and $\beta$. We define a prefix as }p_i \triangleq x_0^i for }0 \leq i \leq N. We determine the lexicographic order of prefixes by comparing symbol to symbol, from high to low indices (the symbol with the highest index in the prefix is most significant), and we use the terms larger and smaller for prefixes and symbols in a lexicographic sense. The prefixes are uniquely ordered because the sentinel symbol is larger lexicographically than any symbol in }X. The problem of prefix sorting can now be defined formally as determining the unique sorted order (in increasing order) of all the prefixes }p_i, }0 \leq i \leq N.

Example 2 Consider }x_1^{11} = 01011111111 from Example 1. After appending the sentinel, }x_0^{11} = $01011111111. The sorted order of the prefixes in increasing order is }p_3 = $010,
\[ p_1 = \$0, p_4 = \$0101, p_2 = \$01, p_5 = \$01011, p_6 = \$010111, p_7 = \$0101111, p_8 = \$01011111, p_9 = \$010111111, p_{10} = \$0101111111, p_{11} = \$01011111111, \text{and } p_0 = \$ . \]

An alternative to prefix sorting is suffix sorting (see Section 5.2). We append a sentinel symbol \$ at the end of the sequence, i.e., \( x_{N+1} = \$ \), and define a suffix as \( s_i \triangleq x_i^{N+1} \) for \( 1 \leq i \leq N \). The problem of suffix sorting can now be defined formally as determining the single sorted order (in increasing order) of all suffixes \( s_i, 1 \leq i \leq N + 1 \).

1.3.3 Burrows Wheeler transform (BWT)

The BWT output, a length-\( N \) sequence denoted \( y \), is obtained by first determining the (unique) sorted order of the prefixes of \( x \); for each prefix \( p_i \) the output is the following symbol \( x_{i+1} \). However, \( p_N \) does not have a following symbol \( x_{N+1} \). Hence, \( x_{N+1} \) is omitted from \( y \); the BWT index that indicates the position of \( x_{N+1} \) in \( y \) is provided as a secondary output instead. The BWT output \( y \), along with the index, suffice for recovering \( x \) [21, 26, 31]. This reconstruction is called the inverse BWT (IBWT); it is a simple algorithm, its computational complexity is \( O(N) \), and it is very fast in practice [31]. The BWT itself is the computational bottleneck in many BWT-based applications and has thus attracted considerable research interest concerning fast implementation [3, 32–35].

**Example 3** In Example 2, \( (p_3,p_1,p_4,p_2,p_5,p_6,p_7,p_8,p_9,p_{10},p_{11},p_0) \) was the sorted order of the prefixes. The symbols following these prefixes are \( (x_4,x_2,x_5,x_3,x_6,x_7,x_8,x_9,\]
\( x_{10},x_{11},x_{12},x_1) \). Since \( x_{12} \), which appears in position 11, is unknown, the BWT index is 11. The BWT output (without the unknown symbol) is \( y = 11101111110 \).

The BWT collects all the symbols following each state \( s \in \mathcal{S} \) together in its input into a continuous segment of its output \( y \), denoted \( \text{segment}_s \). Since each state generates symbols according to some conditional probability, \( y \) is distributed similar to i.i.d. in
each segment, hence the distribution of $y$ is similar to piecewise i.i.d. (PIID) [22, 23, 25].

Owing to this key property, compressing $y$ directly [3, 21–23, 25, 26] often yields good compression results.

**Example 4** According to the sorted prefixes from Example 2, the states that generated $y$ in Example 3 are (0, 0, 01, 01, 11, 11, 11, 11, 11, 11, ?), where “?” is an unknown state (we cannot determine the state that generated $x_1$), and the state 11 that generated $x_{12}$ was removed from this list. Note that the states are arranged in sorted order. Also, $\text{segment}_0 = y_1^2 = 11$, $\text{segment}_{01} = y_3^4 = 10$, $\text{segment}_{11} = y_5^{10} = 111111$, and $\text{segment}_?$, which corresponds to the unknown state, is $y_{11} = 0$. In each segment, the proportions of the symbols are related to the conditional probabilities from Example 1 and Figure 1.1.

### 1.3.4 Prefix and suffix tree methods

In contrast to the suffix tree literature [3, 24, 32, 33, 36–38], which considers suffix sorting, we discuss prefix trees for prefix sorting. In order to simplify the presentation, we refer to both as *prefix tree methods*. Prefix tree constructions play a major role in this dissertation because (i) the $O(N)$ BWT-MDL algorithm (see Chapter 2) relies on their properties, and (ii) to the best of our knowledge, they are the only suffix sorting method whose worst-case complexity is smaller than that of our suffix list algorithms (see Chapter 5).

Following McCreight [37], a *prefix tree $T$* that corresponds to a sequence $x$ (Figure 1.2) contains *internal nodes* and *leaves*. Nodes are connected by *arcs*, where each arc is labeled by some nonempty sequence in $X^*$. Each internal node of $T$, except for the root, has at least two children nodes. Therefore, there are fewer internal nodes than leaves, so $T$ has $O(N)$ nodes. The labels of arcs from an internal node to its children nodes, called *sibling arcs*, differ in their last symbol. Each node corresponds to a sequence that consists of
the appended labels of the arcs on the path from that node to the root. In particular, each leaf corresponds to a prefix.

Prefix trees can also be used to construct the BWT output [3, 24]. By sorting sibling arcs according to their last symbol, the prefixes corresponding to the leaves are lexicographically sorted. Therefore, by maintaining the data structure such that sibling arcs are sorted according to their last symbol, the BWT can be determined by first constructing a prefix tree $T$, and then scanning $T$ in a depth-first search, outputting for each leaf the symbol following its corresponding prefix (c.f. Larsson [3] and Effros [24] for details). Finally, each segment of the BWT output $y$ corresponds to a subtree rooted at one of the nodes of the prefix tree $T$.

**Example 5** The prefix tree $T$ for $x^{11}_0 = \$0101111111$ appears in Figure 1.2. Circles depict internal nodes, squares depict leaves, the arcs have labels, and below each
leaf we describe the corresponding prefix. A depth-first search on \( T \) would scan the leaves from left to right (increasing lexicographical order), corresponding to the prefixes \((p_3, p_1, p_4, p_2, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_0)\). This is identical to the order of the prefixes in Example 2. Note also that each segment from Example 4 corresponds to a subtree rooted at one of the nodes of the prefix tree \( T \).

The prefix tree method suggested by McCreight [37] is nonsequential, i.e., it processes the prefixes from \( p_N \) down to \( p_0 \). When processing \( p_i \), let \( \text{suf}(i) \) denote the longest suffix of \( p_i \) that was also a suffix of some \( p_j \), where \( i < j \leq N \). Define \( \text{start}(i) \triangleq i - |\text{suf}(i)| + 1 \), then \( \text{suf}(i) = x_{\text{start}(i)}^i \). It is necessary to know \( \text{suf}(i) \) because \( p_i \) and \( p_j \) both end with \( \text{suf}(i) \), but \( x_{\text{start}(i)-1} x_{i-|\text{suf}(i)|} \neq x_{j-|\text{suf}(i)|} \); thus, \( \text{suf}(i) \) must become an internal node in \( T \), if it is not one already. If \( \text{suf}(i) = sx_i \) for some sequence \( s \), then \( sx_i \) is a suffix of both \( p_i \) and \( p_j \); hence, \( s \) is a suffix of both \( p_{i-1} \) and \( p_{j-1} \), so \( s \) is a suffix of \( \text{suf}(i-1) \).

Prefix tree construction maintains prefix links that enable going from any internal node corresponding to a sequence of the form \( sa \), where \( s \in \mathcal{X}^* \) and \( a \in \mathcal{X} \), to the node corresponding to \( s \) (the total number of prefix links during the prefix tree construction is \( O(N) \)). Following the prefix link of \( \text{suf}(i) \) to the node corresponding to \( s \), it is possible that \( s = \text{suf}(i-1) \); this can be determined in \( O(1) \) computations per prefix. Alternatively, \( s \) is a suffix of \( \text{suf}(i-1) \), but \( x_{(i-1)-(|s|-1)-1} = x_{\text{start}(i)-1} \) is identical to one of the symbols that precedes \( s \) in the prefix tree. In this case, symbol comparisons on \( x_{\text{start}(i)-1} \), \( x_{\text{start}(i)-2} \), etc., are performed until \( \text{suf}(i-1) \) can be determined; in total this requires \( \text{start}(i) - \text{start}(i-1) + 1 \) symbol comparisons. Therefore, \( N \) prefixes need to be processed and

\[
\sum_{i=1}^{N} \text{start}(i) - \text{start}(i-1) + 1 = N + \text{start}(N) - \text{start}(0) \leq 2N
\]

symbol comparisons need to be performed. For a binary alphabet, maintaining the prefix tree data structure requires \( O(1) \) operations per processed prefix, and \( O(1) \) operations
per symbol comparison, for a total of \( O(N) \) operations. For nonbinary alphabets, the computational complexity of prefix tree methods is still linear in \( N \), but \(|\mathcal{X}|\) affects the multiplicative constant. Recent work relating to prefix tree methods has considered sequential variants \([24,36,38]\) and refinements of the data structure for reducing the memory use \([32]\).

1.3.5 Context trees

A context tree is a data structure that stores information about contexts preceding symbols in the input sequence \( x \). Context trees have been used in a variety of source coding algorithms in the past \([6,8,11]\) and are also related to the basic version of the new BWT-MDL algorithm. In contrast to prefix trees, where labels of arcs are sequences in \( \mathcal{X}^* \), the labels of the arcs of a context tree are symbols in \( \mathcal{X} \). Furthermore, the context tree only stores nodes that appeared in at least two prefixes, i.e., the context tree stores information up to branches that correspond to a single prefix. Giegerich and Kurtz \([36]\) differentiate between atomic prefix trees, wherein each arc is a symbol of \( \mathcal{X} \), and compact prefix trees that only have branching nodes. Note that context trees are similar to atomic prefix trees, whereas the prefix trees that we use are compact.

Unfortunately, context trees may represent the input \( x \) inefficiently, because for every arc label of length \( k \) going to an internal node in a compact prefix tree, the atomic prefix tree has \( k \) nodes (Figure 1.3). In fact, a context tree may have \( O(N^2) \) nodes, as shown by the following construction. Suppose that \( N = 3M + 2 \) for some \( 0 < M \in \mathbb{Z} \). Consider \( x \) such that \( x_i = 1 \) for \( i \in \{M + 1, 2M + 2\} \) and \( x_i = 0 \) elsewhere. For any \( m \in \{1, \ldots, M\} \), there are only two prefixes, \( p_{M+m+1} \) and \( p_{2M+m+2} \), whose last \( m+1 \) symbols are 1 followed by \( m \) appearances of 0. But \( x_{1}^{M+m+1} = x_{M+2}^{2M+m+2} \), so \( p_{M+m+1} \) and \( p_{2M+m+2} \) share their last \( M + m + 1 \) symbols. Therefore, in the context tree for \( x \) there are at least \( M + 1 \) nodes that correspond to contexts that only \( p_{M+m+1} \) and \( p_{2M+m+2} \)
had. Recalling that \( m \in \{1, \ldots, M \} \), the context tree for \( x \) has more than \( M^2 = O(N^2) \) nodes that correspond to at least two prefixes. Our \( O(N) \) BWT-MDL algorithm relies on (compact) prefix trees instead of (atomic) context trees.

### 1.3.6 Context depths

In universal source coding, the maximal context depth \( D \) is unknown. One approach assumes an arbitrary constant bound \( D_{\text{max}} \) on the depth, and considers all tree source models up to this depth, i.e., we must process each node of a context tree whose depth is up to \( D_{\text{max}} \). In contrast, the maximal depth \( D \) of a tree source is a property of its structure \( \mathcal{S} \). Using fixed \( D_{\text{max}} \) may fail to yield good compression if \( D_{\text{max}} < D \). Another approach is to use \( D_{\text{max}}(N) \) that increases with \( N \). Asymptotically, \( D_{\text{max}}(N) \geq D \), and the class of models considered will include the actual tree source \( \mathcal{S} \) that generated \( x \), leading to asymptotic identification of \( \mathcal{S} \). Therefore, with this approach, the expected redundancy bound (1.1) can be achieved asymptotically.

Unfortunately, practical inputs (text, DNA sequences, financial information, etc.) are only approximated by tree sources. In this case, although there is no “actual tree source model,” the question is, for a given \( N \), what \( D_{\text{max}}(N) \) should be used in the search for
the MDL tree source model. Szpankowski [39] analyzed the depths of leaves of atomic prefix trees. We denote the depth of the deepest leaf of a context tree by $D_{\text{deep}}(N)$. Szpankowski proved for stationary ergodic sources that, under some mixing conditions, 
\[
\lim_{N \to \infty} \frac{D_{\text{deep}}(N)}{\log(N)} = \frac{1}{h_2}
\] almost surely, where $h_2$ was specified in closed form [39, p.1650], and $0 < h_2 \leq H$. Kontoyiannis and Suhov [40] considered a weaker mixing condition: following (almost) any semi-infinite past $x^0_{-\infty}$ of a sequence $x$ generated by a random process, there exists an integer $r \geq 1$ and a real number $\omega \in (0,1)$ such that, after $r$ additional symbols have passed, $P(x_r = \alpha|x^0_{-\infty}) \geq \omega > 0$ for any $\alpha \in \mathcal{X}$. When this condition holds,\footnote{This condition appears very mild.} we have \[
\lim_{N \to \infty} \frac{D_{\text{deep}}(N)}{\log(N)} \leq -3r/\log(1-\omega) + \epsilon\] with probability one for any $\epsilon > 0$. Therefore, for stationary ergodic sources that satisfy the mixing condition, $D_{\text{max}} = [-3r/\log(1-\omega) + \epsilon] \log(N)$ suffices to asymptotically guarantee that the entire context tree is processed; hence, the MDL source will be determined and the expected redundancy bound (1.1) will be achieved.

In view of these considerations, we differentiate between source coding algorithms according to the maximal context depths that they can support with worst-case complexity below $O(N^{1+\varepsilon})$ for any $\varepsilon > 0$, i.e., complexity of the form $O(N)$, $O(N \log(N))$, etc. \textit{Increasing depth} algorithms support depths of the form $0 < D_{\text{max}}(N) \leq \sigma^* \log(N)$, for some $\sigma^* > 0$. Increasing depth algorithms asymptotically achieve the expected redundancy bound (1.1) for tree sources, but may not suffice for MDL identification of stationary ergodic sources. Depths as large as any logarithmic growth (LLG), i.e., $D_{\text{max}}(N) \geq \sigma \log(N)$ for any $\sigma > 0$, for $N \geq N(\sigma)$, imply that the MDL source is asymptotically identified for stationary ergodic sources. Therefore, LLG algorithms asymptotically achieve (1.1) for stationary ergodic sources.

However, in some cases there may be highly repetitive structures in $x$. For example, facsimiles (binary images) often have large regions that are entirely white; if $x$ is a
raster scan of such an image, contexts of $O(\sqrt{N})$ depth may be needed. Unbounded depth algorithms support a maximal depth of $D_{\text{max}}(N) = N$; they always identify the MDL source, and hence they achieve both redundancy bounds (1.1) and (1.2). If $x$ was generated by a tree source, supporting unbounded context depths may not be necessary, but it is potentially useful for a general purpose compression algorithm. The BWT-MDL algorithm of Chapter 2 is an unbounded depth encoder; it requires $O(N)$ time to identify the (unbounded depth) MDL source.

1.4 Related Source Coding Methods

In this section, we describe several approaches to source coding. The key issues that we emphasize are the redundancies attained by different algorithms, their computational complexities, and the context depths that they support. In particular, we show that the complexity of previous source coding algorithms that attain the redundancy bounds (1.1) and (1.2) is $O(N \log(N))$. We also identify inherent redundancies in previous uses of the BWT. Overcoming these redundancies will lead to our BWT-MDL algorithm in Chapter 2.

1.4.1 The “plug-in” approach

A classical approach for sequential source coding of tree sources is to estimate for each input symbol $x_i$ what tree source $S$ is optimal according to $x_i^{i-1}$, the portion of the input sequence that has already been processed. In fact, it suffices to estimate for $x_i$ the optimal context depth $\hat{L}_i$ [6, 8, 10, 11], and assign probabilities to $x_i$ according to the frequency of symbols in $x_i^{i-1}$ that followed the context $x_{i-L_i}^{i-1}$ of $x_i$. This “plug-in” approach uses the best context depth from the past in order to assign probabilities in the present. Despite the intuitive appeal, the “plug-in” approach does not attain the pointwise redundancy bound (1.2) for every input [27]. However, the “plug-in” approach
does attain the expected redundancy bound (1.1); the pointwise bound is achieved in expectation and with probability one.

Over the years, research on the “plug-in” approach has concentrated on finding good estimates for $\hat{L}_i$ [6, 8, 10, 11] and developing low complexity algorithms that accumulate information on the input sequence $x$ with context trees [8, 10, 11]. One obvious complexity bottleneck is the computation involved in growing the context tree. Although prefix trees can accumulate this information sequentially [24, 36, 38], less efficient algorithms were used in the past [6, 8, 10, 11]. Another complexity bottleneck, which is sometimes overlooked, is arithmetic coding [1, 10, 41]. In order to approach the redundancy bounds, the arithmetic coder must introduce at most $O(1)$ bits of expected redundancy. This can be achieved by implementing the arithmetic coding using computations performed with $\log(N)$ bits of precision [10]. Finally, when unbounded context depths are considered, the aggregate typical complexity for determining which state generated each of the symbols $x_i$ is $O(N\log(N))$, but the worst-case complexity is at least $O(N^2)$ because context trees may have $O(N^2)$ nodes. The typical complexity is $O(N\log(N))$ because traversing the context tree data structure according to the context of $x_i$ requires an $O(\log(N))$ (expected) excursion through the tree (see Section 1.3.6). To the best of our knowledge, this complexity bottleneck has not received previous research attention. Note that, because the worst-case complexity of these unbounded algorithms is at least $O(N^2)$, according to our criteria these algorithms do not support unbounded depths. However, by limiting $D_{\text{max}}$, there will be at most $O(ND_{\text{max}})$ nodes in the context tree. In this case, some of these algorithms have $O(ND_{\text{max}})$ worst-case complexity, which implies that they are LLG algorithms. In the remainder of this section, we briefly outline several source coding algorithms based on the “plug-in” approach.

Algorithm “context” [6] introduced context trees as a useful tool for source coding and provided a suboptimal rule for choosing $\hat{L}_i$. The redundancy of algorithm “con-
text” does not achieve the redundancy bounds because of the suboptimal context rule. Algorithm “context” requires $O(ND_{avg})$ operations performed with $O(\log(N))$ bits of precision, where $D_{avg}$ is the average depth of leaves of the context tree. Therefore, for stationary ergodic sources, the typical complexity is $O(N\log(N))$. However, the worst-case complexity is at least $O(N^2)$. By limiting $D_{max}$, algorithm “context” can support LLG depths. Weinberger, Lempel, and Ziv [10] provided a Neyman-Pearson criterion for estimating the context $\hat{L}_i$. They also provided an arithmetic coding scheme that introduces $O(1)$ bits of expected redundancy by implementing the arithmetic coding using computations performed with $\log(N)$ bits of precision. They claimed $O(N)$ complexity by using blocks of increasing lengths and estimating $S$ at the beginning of each block. However, because the number of blocks grows with $N$, each estimation step of $S$ must require sublinear complexity. Each estimation step has $O(|\mathcal{X}|^{D_{max}(N)})$ complexity, so $D_{max}(N) \leq \log(|\mathcal{X}|(N));$ hence, this algorithm supports increasing depths. Weinberger, Lempel, and Ziv asymptotically achieve the expected redundancy bound (1.1) for tree sources (see Section 1.3.6). However, they overlooked the fact that the determination of the state that generated each of the symbols requires $O(N\log(N))$ operations. Weinberger, Rissanen, and Feder [11] extended the improved redundancy performance to unbounded context depths. Their complexity is similar to that of “context” [6]; they support LLG depths. Following the arguments of Section 1.3.6, they asymptotically achieve (1.1) for stationary ergodic sources. Rissanen [8] proposed an algorithm based on a novel context tree construction that processes blocks of increasing lengths. Rissanen claimed that the aggregate complexity required for estimating $S$ at the beginning of each of the blocks is $O(N)$. Unfortunately, a careful examination of the block lengths [8, p.1069] reveals that the aggregate complexity for estimating $S$ at the beginning of each of the blocks is actually $O(N \log(N))$. According to our criteria, Rissanen’s algorithm supports
unbounded depths. The algorithm attains the expected redundancy bound (1.1); the pointwise bound (1.2) is achieved in expectation and with probability one.

The “plug-in” approach has also received attention from the practical data compression community. In particular, prediction by partial matching (PPM) is a popular method [24, 42, 43] whose practical compression aspects have been researched extensively [42, 43]. However, its redundancies have received less attention.

1.4.2 The semipredictive approach

The BWT-MDL algorithm of Chapter 2 is an efficient implementation of the semipredictive approach for universal lossless coding that supports unbounded context depths. The parallel compression algorithm of Chapter 4 is also related to this source coding approach. Accordingly, we now describe the concept underlying semipredictive coding, the redundancy performance attained, and some previous work on this topic.

Consider a tree source structure $S$ whose explicit description requires $l_S$ bits, and denote the probability of the input sequence $x$ using the tree source structure $S$ by $p_S(x)$ (see Section 2.2.2). Using $S$, the coding length required for $x$ is $l_S - \log(p_S(x))$. Define the MDL tree source structure $S^*$ as the tree source structure that provides the shortest description of the data, i.e.,

$$S^* \triangleq \arg \min_{S \in \mathcal{C}} \{l_S - \log(p_S(x))\}, \quad (1.3)$$

where $\mathcal{C}$ is the class of tree source models being considered.

The semipredictive approach processes the input $x$ in two phases. Phase I first estimates $S^*$ by $\hat{S}$ and then encodes the structure $\hat{S}$ explicitly. Phase II uses $\hat{S}$ to encode the sequence $x$ sequentially. For each state $s \in \hat{S}$, a Krächek-S-Trofinov (KT) [44] estimator sequentially feeds probability assignments for symbols generated by $s$ to an
arithmetic encoder. The Decoder first determines \( \hat{\mathcal{S}} \), and afterwards uses it to decode \( x \) sequentially.

The semipredictive approach was described by Rissanen [7]. If the actual tree source structure \( \mathcal{S} \) that generated \( x \) belongs to the class \( \mathcal{C} \) being considered during the coding length minimization, then the resulting coding length is at most \( l_{\mathcal{S}} - \log(p_{\mathcal{S}}(x)) \). Therefore, the semipredictive approach can achieve both the redundancy bounds (1.1) and (1.2). However, because all of \( x \) needs to be processed in Phase I before the encoding begins in Phase II, semipredictive source coding algorithms are nonsequential.

Nohre [5] provided an algorithm that estimates the globally optimal MDL tree source structure \( \mathcal{S}^* \) by pruning the context tree with dynamic programming. His algorithm uses the natural code (see also [13]) to describe the estimated structure \( \hat{\mathcal{S}} \) with \( 2|\hat{\mathcal{S}}| - 1 \) bits. The dynamic programming algorithm, combined with a data structure that Nohre proposed, leads to \( O(N + |\mathcal{X}|^{D_{\text{max}}}) \) complexity. Therefore, Nohre’s algorithm is an increasing depth algorithm (see Section 1.3.6). Although Nohre did not analyze the redundancy of his algorithm, and used Laplace probability estimates (whose redundancy is higher near the edges of the probability simplex), it is easy to show that, with KT probability estimates [44], Nohre’s algorithm asymptotically achieves the expected redundancy bound (1.1) for tree sources.

The semipredictive approach was also studied by additional authors. Willems et al. [9, 14] obtained a semipredictive algorithm by modifying CTW [13]; we call their algorithm CTW-MDL. For tree sources up to some maximal depth \( D_{\text{max}} \), CTW-MDL requires \( O(ND_{\text{max}}) \) operations performed with \( O(\log(N)) \) bits of precision. CTW-MDL is an LLG algorithm, and asymptotically achieves (1.1) for stationary ergodic sources. Larsson [3, 4] provided a semipredictive approach for compressing the BWT output. More details about Larsson’s approach will be given in Section 1.4.4.
1.4.3 Implementing mixtures with CTW

The mixture approach simultaneously encodes $x$ according to all tree source models in $C$, by assigning $S$ a weight $2^{-l_S}$; the probability assigned to $x$ is the weighted sum $\sum_{S \in C} 2^{-l_S} p_S(x)$. (Note that the weighted sum is strictly greater than $2^{-l_{S^*}p_{S^*}(x)}$, hence the mixture approach outperforms the semipredictive approach.) If the actual tree source $S$ that generated $x$ belongs to $C$, the probability assigned to $x$ by the mixture approach achieves the redundancy bounds (1.1) and (1.2). Context tree weighting (CTW) [13] implements the bounded depth mixture sequentially with $O(ND_{\text{max}})$ complexity by maintaining a tree data structure that represents all possible states in all bounded depth tree sources. CTW can support LLG depths; it asymptotically achieves (1.1) for stationary ergodic sources. An unbounded variant of CTW [12] also uses a complicated data structure. Its complexity is similar to that of algorithm “context” [6]. Because limiting $D_{\text{max}}$ in unbounded CTW yields basic CTW [13], unbounded CTW can be modified into an LLG algorithm. Finally, because it processes the entire context tree, unbounded CTW achieves both redundancy bounds (1.1) and (1.2).

1.4.4 Source coding with the BWT

In Section 1.3 we described the BWT in detail and explained why its output distribution is similar to PIID [22, 23, 25]. Owing to this similarity, compressing the BWT output $y$ directly [3, 21–23, 25, 26] often yields good compression results, especially when heuristics for practical sources [21, 22, 26] are utilized. However, even if the Kullback-Leibler divergence [1] between $y$ and PIID [25] is neglected, compressing $y$ using a PIID method cannot achieve universal coding redundancies close to the $\frac{|S|(|X|-1)}{2} \log(N)$ bits implied by Rissanen’s bound [2], because PIID methods require additional information about the transitions between segments.
Merhav [45] proved that for a PIID distribution with transitions at deterministic (but unknown) positions, $\log(N)$ bits per transition are needed, in addition to the $\frac{1}{2}\log(N)$ bits per parameter in (1.1) and (1.2). PIID sources can be encoded using weighting schemes that approach Merhav’s bounds with polynomial computational complexity, as proposed by Willems [46]. Shamir et al. [47, 48] suggested improved $O(N)$ methods that asymptotically attain Merhav’s MDL bound for PIID sources.

Whereas the PIID methods [46–48] assume deterministic transition positions, in the BWT output these positions are random variables. The BWT output contains additional side information that can be used to estimate these random variables, and possibly reduce the redundancy. However, even full knowledge of the tree source parameters can only provide estimates of the transition positions up to some $O(\sqrt{N})$ term, hence there is an inherent limitation in compressing $y$ directly. Furthermore, since $y$ is not PIID in a strict sense [25], the precise analysis of compression methods based on the BWT is further complicated.

Larsson [3, 4] provided a semipredictive approach for compressing the BWT output. He estimates the MDL tree source structure by $\hat{S}$, encodes $\hat{S}$ and the segment lengths in $y$, and then encodes $y$, modeling each segment as i.i.d. By using a prefix tree BWT construction, Larsson’s method has $O(N)$ complexity for unbounded context depths. However, Larsson’s encoding method is a PIID method because segment lengths are equivalent to transition positions. Therefore, this approach also has $\log(N)$ extra bits per segment. Finally, Larsson’s algorithm does not perform a global coding length minimization, because (i) his encoding method describes different states in the tree, and the corresponding segment lengths, with nonconstant coding lengths, and (ii) the redundancy of encoding segments with unknown parameters is not accounted for.
Table 1.1 BWT-MDL encoder vs. other source coding algorithms.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Algorithm</th>
<th>Redundancy Bound</th>
<th>Worst-Case Complexity</th>
<th>Supported Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Expected</td>
<td>Pointwise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semipredictive</td>
<td>BWT-MDL</td>
<td>yes</td>
<td>yes</td>
<td>$O(N)$</td>
</tr>
<tr>
<td></td>
<td>CTW-MDL</td>
<td>stationary</td>
<td>no</td>
<td>$O(ND_{max})$</td>
</tr>
<tr>
<td></td>
<td>Nohre</td>
<td>tree sources</td>
<td>no</td>
<td>$O(N +</td>
</tr>
<tr>
<td>Mixture</td>
<td>CTW</td>
<td>stationary</td>
<td>no</td>
<td>$O(ND_{max})$</td>
</tr>
<tr>
<td></td>
<td>U-CTW</td>
<td>yes</td>
<td>yes</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>&quot;Plug-in&quot;</td>
<td>&quot;Context&quot;</td>
<td>no</td>
<td>no</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td></td>
<td>WLZ</td>
<td>tree sources</td>
<td>no</td>
<td>$O(N \log(N))$</td>
</tr>
<tr>
<td></td>
<td>WRF</td>
<td>stationary</td>
<td>no</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td></td>
<td>R99</td>
<td>yes</td>
<td>w.p.1</td>
<td>$O(N \log(N))$</td>
</tr>
</tbody>
</table>

We summarize the attributes of the various source coding methods in Table 1.1.² PIID compression methods are not described in Table 1.1 because they are based on the imprecise intuition that the BWT output distribution is similar to PIID [22, 23, 25], hence PIID methods have relatively high redundancies. Our BWT-MDL algorithm (see Chapter 2) is listed, for comparison, in the top row of the table. This semipredictive algorithm is based on Nohre’s context tree pruning, CTW-MDL, and Larsson’s algorithm. The novelty of BWT-MDL is an $O(N)$ implementation of the semipredictive encoder that achieves the redundancy bounds (1.1) and (1.2) while supporting unbounded context depths. Our parallel algorithm (see Chapter 4) has $O(N/B)$ complexity and it supports a maximal depth $D_{max}(N) \approx \log(N/B)$, hence it is an increasing depth algorithm. The parallel algorithm asymptotically achieves the expected redundancy bound (1.1) for tree sources, but may not suffice for MDL identification of stationary ergodic sources.

²We compare the algorithms BWT-MDL, CTW-MDL [9, 14], Nohre [5], CTW [13], U-CTW (unbounded CTW) [12], “Context” [6], WLZ (Weinberger et al.) [10], WRF (Weinberger et al.) [11], and R99 (Rissanen’s 1999 algorithm) [8]. Unbounded CTW, algorithm “context,” and WRF become LLG by limiting $D_{max}$. By “stationary” and “tree source” we mean that for an input generated by the respective class of sources, these algorithms asymptotically identify the MDL source. Finally, BWT-MDL and unbounded CTW (without some limiting $D_{max}$) identify the MDL source w.r.t. the class C of all tree sources; they achieve both the redundancy bounds (1.1) and (1.2). R99 achieves the expected redundancy bound (1.1); the pointwise bound (1.2) is achieved in expectation and with probability one.
CHAPTER 2

AN $O(N)$ SEMIPREDICTIVE UNIVERSAL ENCODER VIA THE BWT

2.1 Introduction

The main goal of current research is to develop algorithms featuring fast computation and low memory use, while providing compression near (1.1) and (1.2). The recent interest in the Burrows Wheeler transform (BWT) can be understood in this context. The BWT [21] has attracted intense research interest because it achieves compression results near the state of the art while being more efficient in terms of computation and memory use [3,21,22,24,26]. Previous practitioners and researchers ran the BWT on input sequences and compressed the BWT output directly [3,21–23,25,26]. Because the BWT output distribution is similar to piecewise i.i.d. (PIID) [22,23,25], compression results were quite good.

In this chapter, we offer several improvements to the previous state of the art in BWT-based compression and semipredictive encoding. First, we use the BWT as an efficient computational tool for pruning context trees [3–5,9,14], and encode the input sequence rather than the BWT output. Second, we implement the MDL estimation phase of the semipredictive approach by incorporating suffix tree methods [3,24,32,33,36–38] to construct the unbounded depth context tree in $O(N)$ time. Third, we show how back-
tracking from the BWT output to the input sequence enables one to determine which tree
source state generated each of the symbols of the input in $O(N)$ worst-case complexity,
in contrast to previous algorithms that required $O(N \log(N))$ typical (and often $O(N^2)$
worst-case) complexity. Combining these contributions, we have an $O(N)$ nonsequential
semipredictive encoder whose pointwise redundancy w.r.t. any (unbounded depth) tree
source is $O(1)$ bits per state above the redundancy bounds (1.1) and (1.2). Unfortu-
nately, backtracking from the BWT output cannot be performed in the decoder; hence,
we leave the development of low complexity decoders as an important open problem.

Byproducts of the efficient and explicit MDL tree source identification provided by our
algorithm include universal statistical model extraction algorithms, for use in estimation,
classification, and other problems in statistical inference. Possible specific applications
include speech and image recognition, and genome analysis. In terms of impact on
future research directions, we hope that our nonsequential semipredictive algorithm will
motivate work on $O(N)$ algorithms for the “plug-in” and mixture approaches.

The remainder of the chapter is organized as follows. Section 2.2 describes the basic
version of the new BWT-MDL algorithm, Section 2.3 lays out the $O(N)$ version of BWT-
MDL, Section 2.4 discusses some properties of our method, and Section 2.5 concludes
and suggests several topics for further research.

## 2.2 Basic BWT-MDL Algorithm

### 2.2.1 Key idea

In Chapter 1, we explained that Larsson [3, 4] provided a semipredictive approach for
compressing the BWT output $y$. By using a prefix tree BWT construction, Larsson’s
method has $O(N)$ complexity for unbounded context depths. We now explain why com-
pressing $y$ has limited potential in a semipredictive framework, and offer to use the BWT only as a fast computational tool for estimating the MDL tree source structure.

The key realization that leads to our BWT-MDL method is based on the following perspective on the coding length needed for describing the BWT output $y$. The symbols in $\text{segment}_s$ of $y$ are identical to the symbols generated by state $s$ in $x$; hence any probability assignment scheme will assign the same probability to these symbols. Furthermore, $x$ can be recovered from $y$ using the $\log(N)$ bits needed to encode the BWT index, and per (1.2), $x$ can be encoded within $\frac{N}{2}\log(N)$ bits of the ML entropy. Therefore, it makes sense that $y$ can be encoded within $\frac{N+2}{2}\log(N)$ bits of the ML entropy, and there must be some way to avoid the $\log(N)$ bits per transition paid when using PIID methods [45–48].

Because the symbols in $y$ are sorted according to contexts that preceded them, $\text{segment}_{1s}$ follows $\text{segment}_{0s}$ in $y$, and the first position of $\text{segment}_{1s}$ is thus suspect beforehand as a possible transition. In order to determine the transition positions, we must rely on the connection between $x$ and $y$, which is given by the inverse BWT (IBWT) [21, 26, 31]. But if $y$ is encoded sequentially, as in the PIID compression methods [3, 4, 21–23, 25, 26], we cannot utilize the IBWT until all of $y$ has been encoded. Therefore, it seems that, in order to avoid paying $\log(N)$ bits per transition, we should not compress $y$ directly.

We overcome the extra redundancy inherent in Larsson’s method [3, 4] (see Section 1.4.4) by encoding the original input sequence $x$ sequentially according to the estimated tree source structure $\hat{\mathcal{S}}$, instead of encoding the BWT output $y$. In effect, we apply Larsson’s method to the coding strategies of Nohre [5] and CTW-MDL [9, 14]. In contrast to Larsson’s method, which optimizes over imprecise coding lengths and hence does not perform a global coding length minimization, the expressions for the coding lengths of Nohre and CTW-MDL enable us to find a globally minimum coding length. Although BWT-MDL requires precomputing the BWT output $y$, it then processes $x$ se-
quently; hence we avoid coding segment lengths, and close the gap between traditional BWT-based methods and the redundancy bounds (1.1) and (1.2). In the remainder of this section, we describe a BWT-based algorithm that implements Phases I and II of the semipredictive approach with superlinear worst-case complexity; we call this the Basic BWT-MDL algorithm. In Section 2.3, a small but important modification will yield our $O(N)$ BWT-MDL algorithm.

2.2.2 Algorithm overview

We now provide an overview of our semipredictive BWT-MDL algorithm. In the encoder, our goal is to estimate the MDL tree source structure (1.3) and describe the MDL tree structure $S^*$ explicitly in Phase I, and then encode $x$ using $S^*$ in Phase II. In the decoder, our goal is to reconstruct $S^*$ and use it to decode $x$.

**Phase I** - consider a binary alphabet, i.e., $\mathcal{X} = \{0, 1\}$. Recalling that the BWT is an invertible permutation transform, let $V(i)$ be the function that maps every $y_i$ to its corresponding $x_{V(i)}$. The function $V(i)$ is easily computed following the BWT, and thus the preceding symbol in $x$, $x_{V(i)-1}$, is also easily determined. The prefixes preceding symbols in $y$ are sorted, so we can determine the segment $y^m_i$ whose context is 0 with a binary search [29] for the largest $i$ such that $x_{V(i)-1} = 0$. Once we have determined the segment $y^m_i$ whose context is 0, a binary search on $x_{V(i)-2}$ over $i \in \{1, 2, \ldots, m\}$ can determine the segments whose contexts are 00 and 10. We thus investigate the (atomic) context tree of possible segments hierarchically. For each state $s$ we either keep states 1s and 0s in the tree, or merge them into a single state, according to which decision minimizes the coding length; the minimization is described in detail in Section 2.2.3.

We encode the structure $\hat{S}$, the outcome of the minimization, with the natural code [13] (Nohre [5] uses a similar definition). The code $natural_s$ of a subtree rooted at $s \in \mathcal{X}^*$ is empty if $|s| = D_{max}$ ($D_{max}$ represents the maximal allowable depth in the tree, and

26
if \(D_{\text{max}} \geq N\) all possible trees are considered); otherwise, it is 0 if \(s \in \mathcal{S}\), else it is 1 followed by the natural codes of 0s and 1s. The structure \(\hat{\mathcal{S}}\) is described explicitly by the code \(\text{natural}_{\lambda}\) of the entire tree rooted at the empty sequence \(\lambda\). This code is denoted alternatively by \(\text{natural}_{\hat{\mathcal{S}}}\). For a binary alphabet, the tree source has \(|\hat{\mathcal{S}}|\) leaves and \(|\hat{\mathcal{S}}| - 1\) internal nodes, so the length \(|\text{natural}_{\hat{\mathcal{S}}}|\) of the natural code is at most \(2|\hat{\mathcal{S}}| - 1\) bits; this is the model redundancy [13] of BWT-MDL. If \(|\mathcal{X}| > 2\), the relative proportion of leaves is higher, and a more efficient coding would assign leaves a higher probability.

**Example 6** With \(D_{\text{max}} = 2\), the natural code of the tree structure from Example 1 is

\[
\text{natural}_{\mathcal{S}} = \text{natural}_{\lambda} = 1 \text{ natural}_0 \text{ natural}_1 = 1 0 1 \text{ natural}_{01} \text{ natural}_{11} = 101.
\]

Following (1.3), and using the natural code for the explicit description of the tree source structure, i.e., \(l_{\mathcal{S}} = |\text{natural}_{\mathcal{S}}|\), the MDL structure \(\mathcal{S}^*\) is

\[
\mathcal{S}^* \triangleq \arg\min_{\mathcal{S} \in \mathcal{C}} \{|\text{natural}_{\mathcal{S}}| - \log(p_{\mathcal{S}}(x))\}.
\]  

In Section 2.2.3 we will explain why \(\hat{\mathcal{S}}\), the outcome of Phase I, is an MDL tree structure. We thus refer to \(\hat{\mathcal{S}}\) as an MDL tree source structure.

**Phase II** - given the structure \(\hat{\mathcal{S}}\), we encode the sequence \(x\) sequentially. For each symbol \(x_i\), we determine the state \(s \in \mathcal{S}\) that generated \(x_i\) by using the observation that \(y_i \in \text{segment}_s\) implies that \(x_{V(i)}\) was generated by state \(s\). This leads to an \(O(N)\) algorithm for determining which state generated each of the symbols (see Section 2.2.4). This algorithm requires knowing \(B_{\hat{\mathcal{S}}}\), a list of ordered pairs of integers that describes for each \(s \in \hat{\mathcal{S}}\) the boundaries of \(\text{segment}_s\) in \(y\). Each ordered pair consists of the first and
last positions of \( \text{segment}_s \) in \( y \); an empty segment is represented by the ordered pair \((-1, -1)\). Because the boundaries of segments are known during Phase I, we construct \( B_S \) during Phase I.

For each state, we store the symbol counts \( \{n^\alpha_s(x^{i-1}_1)\}_{\alpha \in \mathcal{X}} \), which are the number of times each symbol has been generated by the state \( s \) in \( x^{i-1}_1 \). The probabilities for \( p(x_i = \alpha|^{i-1}_1 x_{\leq i}) = \alpha, \alpha \in \mathcal{X} \) are estimated with a Krichevsky-Trofimov (KT) \cite{KT} estimator as

\[
p\left(x_i = \alpha \right| \{n^\beta_s(x^{i-1}_1)\}_{\beta \in \mathcal{X}, x^{i-1}_1} = s\) = \frac{n^\alpha_s(x^{i-1}_1) + \frac{1}{2}}{|x| + \sum_{\beta \in \mathcal{X}} n^\beta_s(x^{i-1}_1)}, \quad \alpha \in \mathcal{X}. \tag{2.2}
\]

These probabilities are fed into an arithmetic coder \cite{Arim.1, Arim.2, Arim.3}, which encodes \( x \) sequentially.

Let \( n_s(x) \triangleq \sum_{\alpha \in \mathcal{X}} n^\alpha_s(x) \). We define the ML conditional probabilities as \( p^\alpha_s(x) = \frac{n^\alpha_s(x)}{n_s(x)}, \alpha \in \mathcal{X} \), and the ML entropy of symbols generated by \( s \) in \( x \), i.e., the entropy w.r.t. the best i.i.d. model with parameters set to their ML estimates, as \( H_s(x) \triangleq H(\{p^\alpha_s(x)\}_{\alpha \in \mathcal{X}}) \),

\[
H(\{p^\alpha_s(x)\}_{\alpha \in \mathcal{X}}) \triangleq -\sum_{\alpha \in \mathcal{X}} p^\alpha_s(x) \log(p^\alpha_s(x)), \tag{2.3}
\]

and \( 0 \log(0) \) is defined as 0. For i.i.d. sources, the ML entropy lower bounds the achievable compression rate \cite{Arim.1}. The following lemma bounds the redundancy of KT probability estimates\footnote{Previously, we defined redundancy as excess coding length. Here we use the same definition, given the relation \( p = 2^{-l} \) between the probability \( p \) and coding length of \( l \) bits. This definition does not require \( l \) to be an integer.}; this is the parameter redundancy \cite{ParRed} of our method.

**Lemma 1** ([13, 44]) The pointwise redundancy of Krichevsky-Trofimov probability estimates over the ML entropy for a length \( N \) input sequence is \( \frac{|X|-1}{2} \log(N) + O(1) \) bits.

Lemma 1 enables us to upper bound \( |S^*| \). This bound will be useful later.

**Corollary 1** The MDL tree source structure \( S^* \) satisfies \( |S^*| \leq O(N) \).
Proof: Consider $\mathcal{S} = \{\lambda\}$, the tree source structure that has a single state at the root, which is an i.i.d. model. Lemma 1 implies that

$$-\log(p_S(x)) \leq N \log(|\mathcal{X}|) + \frac{|\mathcal{X}| - 1}{2} \log(N) + O(1).$$

Because $|\text{natural}_{\mathcal{S}}| = 1$ and $|\text{natural}_{\mathcal{S}^*}| - \log(p_{\mathcal{S}^*}(x)) \leq |\text{natural}_{\mathcal{S}}| - \log(p_S(x))$, we have

$$|\text{natural}_{\mathcal{S}^*}| - \log(p_{\mathcal{S}^*}(x)) \leq N \log(|\mathcal{X}|) + \frac{|\mathcal{X}| - 1}{2} \log(N) + O(1).$$

The result follows because, for any $\mathcal{S}$, $|\text{natural}_{\mathcal{S}}| = O(|\mathcal{S}|)$. \qed

**Decoder** - The BWT-MDL decoder is similar to the decoders of Nohre [5] and CTW-MDL [9, 14]. The decoder first determines $\hat{S}$ from the natural code, and afterwards uses it to decode $x$ sequentially. The decoder maintains symbol counts, and can thus determine for each symbol what its KT probabilities in Phase II of the encoder were. As explained before, determining which state generated each of the symbols of an input sequence can be $O(N \log(N))$. Although we have provided an $O(N)$ solution to this problem in the BWT-MDL encoder, we have yet to develop an $O(N)$ complexity decoder. We leave this important problem for future research, and concentrate on the encoder in the remainder of the chapter.

**Bounded and unbounded depths** - The algorithm can be used in two variants. The *bounded* variant considers all tree sources whose depth is up to $D_{\max}$. In contrast, the *unbounded* variant considers all possible tree sources of any depth. Pseudocode for the Basic BWT-MDL encoder appears in Figure 2.1. We now describe the details of Phase I, including the bounded depth hierarchical partitioning MDL routine, whose pseudocode appears in Figure 2.2 on page 33.

### 2.2.3 Phase I

Recall that the symbols in $\text{segment}_s$ of $y$ are identical to the symbols generated by state $s$ in $x$. Therefore, merging states $1_s$ and $0_s$ in the context tree for $x$ [5] is
1. Compute the BWT

2. **Phase I**: estimate the MDL tree source structure $\hat{S}$ by running $MDL(\lambda, 1, N)$, and encode $\hat{S}$ with the natural code

3. **Phase II**: use $\hat{S}$ to encode $x$ sequentially

**Figure 2.1** Pseudocode for the Basic BWT-MDL encoder. The bounded depth $MDL$ routine is described in Figure 2.2.

Equivalent to merging $segment_0$s and $segment_1$s of $y$ into $segment_s$. Therefore, instead of considering the pruning of a context tree, as proposed by Nohre [5], we discuss the hierarchical partitioning of the segments of $y$.

The decision whether to merge $segment_0$s and $segment_1$s into $segment_s$ is intended to minimize the coding length. With (2.2), the KT probability estimate assigned to symbols of $segment_s$ with symbol counts of $\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}$ is

$$p(\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}) = \frac{\prod_{\alpha \in \mathcal{X}} \prod_{i=0}^{n_s^\alpha(x)-1} \left( i + \frac{1}{2} \right)}{\prod_{i=0}^{n_s^\alpha(x)} \left( i + \frac{|\mathcal{X}|}{2} \right)} = \frac{\prod_{\alpha \in \mathcal{X}} \Gamma \left( \frac{|\mathcal{X}|}{2} + n_s^\alpha(x) \right)}{\Gamma \left( \frac{|\mathcal{X}|}{2} + \sum_{\alpha \in \mathcal{X}} n_s^\alpha(x) \right)},$$

where $\Gamma(\cdot)$ is the gamma function. Using the KT estimate [44], the coding length required to describe the symbols of $segment_s$, denoted by $KT_s$, is

$$KT_s = -\log(p(\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}));$$

hence the computation of $KT_s$ only requires additions, subtractions, and gamma function logarithms. We discuss the computation of gamma function logarithms in detail in Section 2.3.1.

We process $segment_s$ after processing segments of the form $segment_{\beta}s$, $\forall \beta \in \mathcal{X}$. These will be called the *children segments* of $segment_s$. For each $segment_{\beta}s$, we compute $MDL_{\beta}s$, the coding length required for describing the symbols of $segment_{\beta}s$ when using
the subtree structure that yields the minimal description length. This MDL coding length for segment $s$ can be determined by comparing the MDL coding lengths of all the children segments to $KT_s$, the KT coding length for segment $s$ (2.4.2.5). This comparison must take into account the following factors.

- **Encoding the tree structure** - we encode the tree structure $\hat{S}$ with the natural code [5, 13], as explained in Section 2.2.2.

- **Coding without knowledge of $x_{1-D_{\max}}^0$** - when processing segment $s$, we hierarchically process its children segments first. However, if $s = x_{1}^{\mid s \mid}$, then $s$ appears as a context. In this case, if $s$ is an internal node in the tree source (i.e., not a state), the symbol $x_{\mid s \mid + 1}$ is preceded by an unknown context. This unknown context appears once; hence we encode $x_{\mid s \mid + 1}$ directly, using $\log(|\mathcal{X}|)$ bits. If $x_{1}^{\mid s \mid} = s$, we also increment $n_{\mid s \mid + 1}^s(x)$ by one.

Taking these considerations into account, we hierarchically compute the symbol counts $\{n_{s}^\alpha(x)\}_{\alpha \in \mathcal{X}}$ and the MDL coding length $MDL_s$ by first hierarchically computing the symbol counts and coding lengths of its children segments. Then, the symbol counts $\{n_{s}^\alpha(x)\}_{\alpha \in \mathcal{X}}$ for segment $s$ are computed recursively as

$$n_{s}^\alpha(x) = 1_{\{x_{\mid s \mid} = s\}} \cdot 1_{\{x_{\mid s \mid + 1} = \alpha\}} + \sum_{\beta \in \mathcal{X}} n_{\beta s}^\alpha(x), \quad \forall \alpha \in \mathcal{X},$$

where $1_{\{\cdot\}}$ are indicator functions, and the $1_{\{x_{\mid s \mid} = s\}} \cdot 1_{\{x_{\mid s \mid + 1} = \alpha\}}$ term accounts for the case when $x_{\mid s \mid} = s$. Note that, in the bounded variant, if $\mid s \mid = D_{\max}$, then $s$ is considered a leaf in the tree source, and we compute $\{n_{s}^\alpha(x)\}_{\alpha \in \mathcal{X}}$ by simply counting symbol occurrences in segment $s$ of $y$. In the unbounded variant, we partition segments further and further until length-1 segments are reached, and determine $\{n_{s}^\alpha(x)\}_{\alpha \in \mathcal{X}}$ accordingly. Finally, $MDL_s$
is computed as
\[
MDL_s = \begin{cases} 
KT_s & \text{if } |s| = D_{\text{max}} \\
1 + \min \left\{ 1_{x_1^{|s|} = s} \cdot \log(|\mathcal{X}|) + \sum_{\beta \in \mathcal{X}} MDL_{\beta s}, \quad KT_s \right\} & \text{otherwise}
\end{cases}
\]
(2.6)

In terms of the natural code, if \( |s| = D_{\text{max}} \), \( s \) is a leaf of the tree, and its natural code is empty, else \( |s| < D_{\text{max}} \), and the natural code requires 1 bit to encode whether \( s \in \mathcal{S} \). The segment itself is either encoded by keeping the children segments (this requires a coding length of \( \sum_{\beta \in \mathcal{X}} MDL_{\beta s} \) bits, and an additional \( \log(|\mathcal{X}|) \) bits if \( x_1^{|s|} = s \), or merging the children segments (equivalent to pruning the context tree) into segment_s with coding length \( KT_s \). When \( |s| = D_{\text{max}} \), the possibility of splitting segment_s into children segments is not considered, hence the coding length is always \( KT_s \). (Nohre’s formulation \cite[3.20]{Nohre}) is similar to (2.6), but encodes depth-\( D_{\text{max}} \) leaves with a 0, and assumes that all depth-\( D_{\text{max}} \) contexts of all symbols are known.)

Phase I in Basic BWT-MDL determines the MDL tree source structure \( \hat{S} \), the resulting coding length, and the natural code for \( \hat{S} \), by computing MDL_{\lambda}, where \( \lambda \) is the empty sequence that corresponds to all of the BWT output \( y \). The outcome of Phase I is an MDL structure because context tree pruning can be shown to be a form of a dynamic programming algorithm for coding length minimization (see Appendix A and Nohre \cite{Nohre} for details). Basic BWT-MDL recursively merges segments just as Nohre \cite{Nohre} prunes the context tree.

Pseudocode for the bounded depth hierarchical partitioning MDL routine appears in Figure 2.2. Its input is segment_s that corresponds to \( y_\alpha^b \). Its outputs are: MDL_s; \{n_s^\alpha(x)\}_{x \in \mathcal{X}}; \text{natural}_s, the natural code for the MDL subtree that best describes segment_s; and \( B_s \), a list of boundaries of segments in the MDL subtree, where the boundaries are represented by ordered pairs of integers (see Section 2.2.2). Since Phase I computes MDL_{\lambda}, it outputs the natural code for \( \hat{S} \), which is then encoded. The flag \( \kappa \) indicates
1. **routine** $MDL(s,a,b)$
2. if ($|s| = D_{max}$) {
3. Determine $\{n_{a}^{o}(x)\}_{x \in \mathcal{X}}$ by counting symbol occurrences in $y_{a}^{o}$
4. Compute $KT_{s}$ using (2.4) and (2.5)
5. $MDL_{s} \leftarrow KT_{s}$
6. $natural_{s} \leftarrow \lambda$ /* empty natural code */
7. $B_{s} \leftarrow (a,b)$
8. } else {
9. $\kappa \leftarrow 1_{\{\forall \{s\}=|s|+1\}}$, $\mu \leftarrow 1_{\{|s|=D_{max}-1\}}$
10. Find $m$ with a binary search for the largest $a \leq i \leq b$ such that $x_{\forall \{s\}=|s|-1} = 0$
11. if ($m \geq a$) {
12. Compute $MDL_{0s}$, $\{n_{0}^{o}(x)\}_{x \in \mathcal{X}}$, $natural_{0s}$, and $B_{0s}$ by running $MDL(0s,a,m)$
13. } else { /* empty segment */
14. $MDL_{0s} \leftarrow 1-\mu$, $\{n_{0}^{o}(x)\}_{x \in \mathcal{X}} \leftarrow (0,0)$, $natural_{0s} \leftarrow \{\lambda$ if $\mu = 1$
15. $B_{0s} \leftarrow (-1,-1)$
16. } /* empty segment */
17. Compute $MDL_{1s}$, $\{n_{1}^{o}(x)\}_{x \in \mathcal{X}}$, $natural_{1s}$, and $B_{1s}$ by running $MDL(1s,m+1,b-\kappa)$
18. } else { /* empty segment */
19. $MDL_{1s} \leftarrow 1-\mu$, $\{n_{1}^{o}(x)\}_{x \in \mathcal{X}} \leftarrow (0,0)$, $natural_{1s} \leftarrow \{\lambda$ if $\mu = 1$
20. $B_{1s} \leftarrow (-1,-1)$
21. $n_{0}^{o}(x) \leftarrow n_{0}^{o}(x) + n_{0}^{o}(x)$, $n_{1}^{o}(x) \leftarrow n_{1}^{o}(x) + n_{1}^{o}(x)$
22. Compute $KT_{s}$ using (2.4) and (2.5)
23. if ($\kappa + MDL_{0s} + MDL_{1s} < KT_{s}$) { /* keep both states */
24. $MDL_{s} \leftarrow \kappa + 1 + MDL_{0s} + MDL_{1s}$
25. $natural_{s} \leftarrow 1$ $natural_{0s}$ $natural_{1s}$
26. $B_{s} \leftarrow B_{0s}B_{1s}$
27. } else { /* merge states (prune context tree) */
28. $MDL_{s} \leftarrow 1 + KT_{s}$
29. $natural_{s} \leftarrow 0$
30. $B_{s} \leftarrow (a,b)$
31. } /* merge states */
32. } /* else */
33. } /* if */
34. } /* routine */

**Figure 2.2** Pseudocode for the bounded depth $MDL$ routine. We use a binary alphabet to keep things simple.

the event that some symbol has $s$ as its context, whereas $\mu$ indicates the event that $|s| = D_{max} - 1$, hence the natural codes of its children nodes are empty. The entire Basic BWT-MDL encoder is outlined in Figure 2.1. An example of the execution of the $MDL$ routine appears in Appendix B.

An unbounded depth routine requires modifying the condition in Line 2 to check whether $|segment_{s}| = 1$, i.e., $a = b$. Furthermore, Lines 3-6 are modified. In particular, we only need to examine $y_{a}$ in order to determine the symbol counts, the KT coding
2. if \(|s| = 1\) \{ /* a = b */
3. Determine \(\{n^s_a(x)\}_{a \in \mathcal{X}}\) by examining \(y_a\)
4. \(KT_s \leftarrow \log(|\mathcal{X}|) /* \text{there is one symbol} */
5. \(MDL_s \leftarrow KT_s\)
6. \(natural_s \leftarrow 0 /* s \text{ is a leaf} */
7. \(B_s \leftarrow (a, a)\)

**Figure 2.3** Pseudocode portion for the unbounded variant of the \(MDL\) routine.

length is \(\log(|\mathcal{X}|)\) bits, and \(natural_s \leftarrow 0\). Pseudocode for the portion of the unbounded MDL routine that differs from the bounded variant appears in Figure 2.3.

### 2.2.4 Phase II

We have already discussed updating the symbol counts and computing the KT probability estimate in Section 2.2.2. We also mentioned that arithmetic coding needs to be performed with \(\log(N)\) bits of precision [10]. We now describe an \(O(N)\) algorithm for determining which state generated each of the symbols (recall from Section 1.4.1 that previous algorithms required \(O(N \log(N))\) complexity for this stage). The algorithm uses two data structures (i) \(B_{\mathcal{S}}\) is a list of ordered pairs of integers that describes the boundaries of \(\text{segment}_s\) in \(y\) for each \(s \in \hat{\mathcal{S}}\), and (ii) \(U\) is an array that describes for each \(x_i\) which state generated \(x_i\). The key observations used in the algorithm are (i) because segment boundaries are known, for each \(y_i\) we can track what segment is being processed, and (ii) \(y_i \in \text{segment}_s\) implies that \(x_{V(i)}\) was generated by state \(s\).

The algorithm first initializes entries of \(U\) with dummy values. Then, for each ordered pair in \(B_{\mathcal{S}}\) that represents the boundaries of \(\text{segment}_s\), we scan \(\text{segment}_s\) in \(y\). For each \(y_i \in \text{segment}_s\), we assign the state \(s\) to \(U_{V(i)}\). The algorithm is \(O(N)\) because the sum of lengths of all segments in \(y\) is upper bounded by \(N\).
After running the algorithm, for each symbol $x_i$ that was generated by some state, $U_i$ indicates which state generated the symbol. Alternatively, if $x_i$ was preceded by an unknown context, $U_i$ still has the dummy value, and we will encode $x_i$ directly. We update the symbol counts, compute the KT probability estimates, and run the arithmetic encoder accordingly.

### 2.3 Linear Complexity Algorithm with Finite Numerical Precision

We first discuss a finite precision scheme for approximating the KT coding length in Section 2.3.1, and prove that with this scheme Phase I approximates the coding length within $O(1)$ of the true optimum. Then, Section 2.3.2 discusses the computational complexity of Basic BWT-MDL and shows that the worst-case complexity of Phase I is superlinear. Section 2.3.3 presents a modified $O(N)$ algorithm based on Larsson’s method [3, 4], which we call the BWT-MDL algorithm. Section 2.3.4 lays out the memory use of BWT-MDL.

#### 2.3.1 Approximating the KT coding length

In Section 2.2.3, we noted that the computation of the MDL coding length requires calculating gamma function logarithms. However, these calculations are performed at a finite numerical precision. Because the pointwise redundancy of BWT-MDL is $O(1)$ bits per state above (1.1) and (1.2) (see Section 2.4.1), it is reasonable to require that the MDL coding length be computed within $O(1)$ bits of the true optimum. This requires calculating the gamma function logarithms at a high precision, as we discuss later.

In order to simplify assessment of the precision and complexity of these computations, our computational model makes the following assumptions (i) length-$N$ sequences are processed on a $k = \log(N)$ bit computer architecture, (ii) $k$ bit words are viewed as
$O(1)$ memory, and (iii) arithmetic computations performed with $O(k)$ bits of precision are atomic operations, i.e., they require $O(1)$ time. This computational model is consistent with current typical 32 and 64 bit computer architectures. Furthermore, access to $O(N)$ memory requires pointer arithmetic performed with $O(k)$ bits of precision, so making a distinction between "simple" operations (e.g., memory access) and "complicated" operations (e.g., divisions and logarithms) would be artificial.

In addition to being accurate, the computation must be fast. In our case, gamma function logarithms can be computed using logarithms of factorials. The straightforward approach would compute $\log(n!)$ by adding up $\log(1), \log(2), \ldots, \log(n)$; with our computational model, this requires $O(n)$ computation. Because a length-$n$ segment thus requires $O(n)$ computation, the aggregate computation over all the segments (corresponding to all nodes, including internal nodes, of the context tree) is

$$\sum_s O(|\text{segments}_s|) = \sum_l \sum_{s \in \mathcal{A}^l} O(|\text{segments}_s|)$$

$$= \sum_l O(N), \quad (2.7)$$

where in (2.7) segments are partitioned according to their lengths, and (2.8) follows because $\sum_{s \in \mathcal{A}^l} |\text{segments}_s| = N - l$. Finally, because the range of $l$ increases with $N$ (see Section 1.3.6), the straightforward approach has superlinear complexity. Instead, we provide a scheme for approximating gamma function logarithms that suffices for computing the MDL coding length within $O(1)$ bits of the true optimum, while requiring only $O(1)$ computation and using $O(1)$ memory per gamma function logarithm.

Using (2.4) and (2.5), we need to compute $\log(\Gamma(t))$ for $t \in \{\frac{1}{2}, 1, \frac{3}{2}, 2, \ldots, \frac{\sqrt{N}}{2} + N\}$. For a length-$N$ input sequence, we precompute these values for $t < \sqrt[N]{N}$ to an absolute precision of $O(\frac{1}{N})$, and store them in a lookup table. To get $O(\frac{1}{\sqrt{N}})$ absolute precision on coding lengths as large as $O(N)$, we need $\log(N^2) + O(1) = O(\log(N))$ bits per entry. For small $t$ we use the lookup table.
For large $t$, i.e., $t \geq \sqrt[3]{N}$, we approximate the gamma function with a form of Stirling’s formula,

$$
\log \left( \hat{\Gamma}(t) \right) = \frac{1}{2} \log(2\pi) + \left( t - \frac{1}{2} \right) \log(t) - \left( t - \frac{1}{12t} \right) \log(e). \quad (2.9)
$$

According to Gordon [49],

$$
\frac{1}{12t} - \frac{1}{360t^3} < \ln \left( \frac{\Gamma(t)}{\sqrt{2\pi t^{-t \cdot \frac{1}{2} e^{-t}}} \ln(t)} \right) < \frac{1}{12t} - \frac{1}{360(t + \frac{1}{6})^3}. 
$$

It follows that the accuracy of (2.9) is

$$
0 < \frac{\log(e)}{360(t + \frac{1}{6})^3} < \log \left( \hat{\Gamma}(t) \right) - \log (\Gamma(t)) < \frac{\log(e)}{360t^3}. \quad (2.10)
$$

For large $t$ the error is at most $\frac{\log(e)}{360N}$.

**Theorem 1** When the computation is performed at $O\left(\frac{1}{N}\right)$ (absolute) precision, Phase I approximates the MDL coding length for the entire input sequence $x$ within $O(1)$ bits of the true optimum.

**Proof:** We first prove that, when evaluating the coding length w.r.t. $S^*$, the aggregate error is $O(1)$ bits. We then prove that the aggregate error w.r.t. the coding length for $\hat{S}$ is also $O(1)$ bits; hence, the estimated coding length must be within $O(1)$ bits of the true MDL coding length.

Using Corollary 1, $|S^*| \leq O(N)$. Since the approximated coding length for $S^*$ adds up the coding lengths for all the states, it adds up at most $O(N)$ coding lengths. Each of these coding lengths, computed with (2.4) and (2.5), consists of $O(1)$ gamma function logarithms; hence at most $O(N)$ terms affect the computation. The error in each coding length when using the lookup table is $O(\frac{1}{N})$. Using (2.10), the error when approximating $\log(\Gamma(t))$ with (2.9) is at most $\frac{\log(e)}{360N}$. With at most $O(N)$ terms added up with $O\left(\frac{1}{N}\right)$ (absolute) precision, the aggregate error is bounded by $O(1)$. 

37
Phase I may process $O(N^2)$ nodes in the context tree for $x$ (see Section 1.3.5), so the aggregate coding length error may be $O(N)$. But with $O(N)$ coding length error, the coding length with $\hat{S}$ is $O(N)$ bits; hence $|\hat{S}| \leq O(N)$, the aggregate coding length error with $\hat{S}$ is $O(1)$ bits, and the error w.r.t. $S^*$ is $O(1)$ bits. □

With our computational model, approximating $\log(\Gamma(t))$ with (2.9) requires computations performed with $O(\log(N))$ bits of precision. These computations are atomic operations; each requires $O(1)$ computation and uses $O(1)$ memory. Accessing $\log(\Gamma(t))$ in the lookup table also requires $O(1)$ computation. Furthermore, constructing the lookup table requires $O(\sqrt[3]{N})$ memory and computation.

### 2.3.2 Computational complexity of Basic BWT-MDL algorithm

#### 2.3.2.1 BWT

The BWT can be computed with prefix tree methods [3, 24, 32, 33, 36–38] whose worst-case complexity is $O(N)$. Alternatively, other methods for computing the BWT [34, 35] with superlinear worst-case complexity can be used.

#### 2.3.2.2 Phase I

Using our computational model and the gamma function approximations of Section 2.3.1, the only “hot-spots” in the MDL routine (Figure 2.2) that might require more than $O(1)$ computation per function call are Lines 3 and 10.

- **Line 3** - counting symbol occurrences in $y^k_a$ is $O(b - a)$, it is executed only for leaf segments, and the aggregate length of all leaf segments is at most $N$. Therefore, the aggregate complexity for Line 3 is $O(N)$.

- **Line 10** - binary searches [29] are used to partition segments. The binary searches require $O(\log(N))$ operations per function call performed with $O(\log(N))$ bits of
precision. Therefore, even if Phase I is limited to $O(N)$ function calls, Line 10 needs to be improved.

Another problem with Basic BWT-MDL is that there may be $O(N^2)$ function calls to the unbounded variant of the MDL routine, because each function call processes a node of the context tree for $x$. We conclude with the following theorem.

**Theorem 2** The worst-case complexity of Phase I of the unbounded variant of the Basic BWT-MDL algorithm is $O(N^2 \log(N))$ computations performed with $O(\log(N))$ bits of precision.

Recall that the complexity of Nohre’s algorithm [5] is $O(N + |\mathcal{X}|^{D_{\text{max}}})$ and the complexity of CTW-MDL [9, 14] is $O(ND_{\text{max}})$. Because a context tree has at most $O(N \log(N))$ nodes for typical $x$ (see Section 1.3.6), and the complexity of Basic BWT-MDL is proportional to the number of nodes in the context tree, the typical complexity of Basic BWT-MDL is similar to that of CTW-MDL, and better than that of Nohre’s algorithm for $D_{\text{max}} \gtrapprox \log_{|\mathcal{X}|}(N)$. Furthermore, we showed that BWT-MDL can be implemented with finite numerical precision.

### 2.3.2.3 Phase II

In Phase II, for each symbol $x_i$ we need to determine the state $s \in \mathcal{S}$ that generated $x_i$, compute the KT probability estimate (2.2), update the symbol counts $n^s_i(x_i^{i-1})$ and $n_s(x_1^{i-1})$, and feed the probability to an arithmetic coder [1, 10, 41].

We determine which state generated each of the symbols using the algorithm of Section 2.2.4, whose complexity was shown to be $O(N)$. The remainder of Phase II is also $O(N)$. When processing $x_i$, the computation of the KT probability estimate (2.2) requires the division of two integers. Updating the symbol counts $n^s_i(x_i^{i-1})$ and $n_s(x_1^{i-1})$ requires the incrementation of integers. Because $n^s_i(x_i^{i-1})$ and $n_s(x_1^{i-1})$ are stored with
\(O(\log(N))\) bits of precision, these computations are \(O(1)\) with our computational model. Finally, arithmetic coding needs to be performed with \(\log(N)\) bits of precision [10]. We conclude with the following theorem.

**Theorem 3** The complexity of Phase II is \(O(N)\) computations performed with \(O(\log(N))\) bits of precision.

### 2.3.3 The linear complexity BWT-MDL algorithm

The MDL routine of Phase I of the Basic BWT-MDL algorithm performs a depth-first scan of the context tree for \(x\). Because a context tree may have \(O(N^2)\) nodes, Basic BWT-MDL may require \(O(N^2)\) function calls. In this section, we will show how the properties of compact prefix trees, which were described in Section 1.3.4, can be used to achieve the \(O(N)\) complexity of Larsson’s method in Phase I [3, 4]. We will perform a depth-first scan of the (compact) prefix tree \(T\) for \(x\); our BWT-MDL algorithm requires \(O(N)\) function calls, each of which requires \(O(1)\) computation.

Prefix trees have \(O(N)\) nodes because all internal nodes, except possibly for the root, are branching. Each leaf of \(T\) corresponds to a prefix; the symbol following the corresponding prefix is a length-1 segment in \(y\). Similarly, each internal node \(t\) of \(T\) corresponds to a sequence \(s\) that consists of the appended labels of the arcs on the path from \(t\) to the root. It can be shown that \(\text{segment}_s\) consists of the length-1 segments that correspond to the leaves of the subtree rooted at \(t\). The relation between segments of \(y\) and nodes of the prefix tree \(T\) is illustrated in Figure 2.4.

Recall that Basic BWT-MDL required \(O(N \log(N))\) complexity to partition \(\text{segment}_s\) into \(\text{segment}_{0s}\) and \(\text{segment}_{1s}\) because binary searches were used (Line 10 of Figure 2.2). However, internal node \(t\) has at least two children nodes, and we can run the MDL routine recursively on the segments that correspond to the children nodes of \(t\); partitioning \(t\) consists of simply following the arc labels to its children nodes. Furthermore, \(y\) is
constructed after $T$; by storing for each node of $T$ an ordered pair of integers that represents the boundaries of the corresponding segment in $y$, the location in $y$ of each of the children segments can be determined easily.

We first construct the prefix tree $T$, the BWT output $y$, and the array $V$. We then invoke the MDL routine on the root node of $T$, just as Basic BWT-MDL runs the MDL routine on $y = \text{segment}_\lambda$, which corresponds to the root node. The MDL routine performs a depth-first scan of $T$. In Line 10, we determine the children nodes of internal node $t$; this requires $O(1)$ computation. Suppose that $t'$ is the father node of $t$. Consider the arc from $t'$, which corresponds to $\text{segment}_{s'}$, to $t$, which corresponds to $\text{segment}_s$, where $s'$ is a suffix of $s$. The arc from $t'$ to $t$ is represented by an ordered pair $(t_1, t_2)$ of indices into $x$ [36], where $x_{t_2}$ is the arc label, and $x_{t_1}^{t_1+|s|-1} = s$ is the first appearance of $s$ in $x$.

The decision whether to merge the children segments of $\text{segment}_s$ (prune $T$ at $t$) must take the following factors into account.
Figure 2.5 The structure of $t'$, $\bar{t}$, $t$, and children nodes of $t$ shows that keeping the children states of $t$ requires keeping the arc from $t'$ to $t$, while pruning the tree requires keeping the length-1 arc from $t'$ to $\bar{t}$. The illustration uses an atomic structure to show the location of “invisible” nodes.

- **Keeping children segments** - if we keep the children segments of $segment_s$, then the natural code for the MDL subtree that best describes $segment_s$ must describe the arc $(t_1, t_2)$ from $t'$ to $t$. Because all leaves that branch from this arc do not correspond to any prefixes, we describe each of those leaves with a single bit 0; we also describe each of the (atomic) internal nodes along the arc with a single bit 1. The structure of $t'$, $t$, and children nodes of $t$ is illustrated in Figure 2.5. For a length-$(t_2 - t_1 + 1)$ arc, the length of the portion of the natural code that describes the arc is $|\mathcal{X}|(t_2 - t_1) + 1$ bits.

- **Merging children segments** - if we merge the children segments of $segment_s$ (prune $T$ at $t$), we can represent the symbols of $segment_s$ with a state $\bar{s}$ that corresponds to a node $\bar{t}$, such that $\bar{s}$ is a suffix of $s$ and $|\bar{s}| = |s'| + 1$. The relations between $t'$, $\bar{t}$, and $t$ are illustrated in Figure 2.5. The portion of natural code that describes $\bar{s}$ is a single bit 0.
• **Leaves** - if \( t \) is a leaf of \( T \), then \( MDL_s \) requires 1 bit for describing \( \bar{s} \) and \( \log(|\mathcal{X}|) \) bits for describing the (length-1) **segments** with a KT probability estimate.

• **Symbols with unknown context** - is \( s = x_{[s]} \) and \( s \) appears elsewhere in \( x \), then there is an internal node \( t \) that corresponds to \( s \); hence the mechanism used for the (possible) direct encoding of \( x_{[s]+1} \) in Basic BWT-MDL applies here, too.

Combining these issues, \( MDL_s \) is computed as

\[
MDL_s = \begin{cases} 
1 + \log(|\mathcal{X}|) & \text{if } t \text{ is a leaf} \\
\min \left\{ |\mathcal{X}|(t_2 - t_1) + 1 + 1 \{z_{t_1=\bar{s}}\} \cdot \log(|\mathcal{X}|) + \sum_{\beta \in \mathcal{X}} MDL_{\beta s}, 1 + KT_s \right\} & \text{otherwise}
\end{cases}
\]

where \( MDL_{\beta s} \) denotes the MDL coding length required for the segment that corresponds to the child node of \( t \) whose incoming arc has \( \beta \) as its last symbol.

Constructing a length \( |\mathcal{X}|(t_2 - t_1) + 1 \) natural code portion when the \( O(N) \) version of the MDL routine processes **segments** would have superlinear worst-case complexity, because the sum of the lengths of the arcs that we process is similar to the number of function calls to the MDL routine performed by Basic BWT-MDL. Therefore, instead of returning natural, the MDL routine returns a set of ordered pairs of integers that represent arcs of the estimated MDL tree source structure, where a leaf is represented by \((-1, -1)\). After the MDL routine finishes, we construct natural using these ordered pairs. Because \(|\hat{S}| \leq O(N)\) (see the proof of Theorem 1), this construction requires at most \( O(N) \) computation.

In summary, there are \( O(N) \) function calls to the MDL routine, and each individual function call requires \( O(1) \) computation; hence, running the MDL routine is \( O(N) \), and constructing natural is \( O(N) \). Recalling that Phase II is \( O(N) \), we conclude with the following theorem.

**Theorem 4** Using prefix tree methods, the BWT-MDL encoder requires \( O(N) \) computations performed with \( O(\log(N)) \) bits of precision.
Remark 1  Theorem 4 applies to both the bounded and unbounded variants.

Remark 2  With prefix trees, Phase I does not need the boundaries of the segments in $y$, because symbol counts are computed hierarchically. We explained how to obtain the boundaries in order to make the analogy to Line 10 more clear, and to construct $B_S$, which is used in Phase II.

In practice, prefix tree methods are often slower than other methods for computing the BWT [34, 35]. Although our $O(N)$ BWT-MDL algorithm is theoretically superior, the Basic BWT-MDL algorithm might be better in practice. Furthermore, the hierarchical structure of segments is often relatively balanced. In this case, the majority of segments are short, and the aggregate complexity with binary searches is also $O(N)$.

2.3.4  Memory use

Computing the BWT uses $O(N)$ memory [3, 24, 32–38].

Phase I performs a recursion into the prefix tree [3, 4]. When the recursion processes segments, it stores $MDL_{\beta s}$, $natural_{\beta s}$, and $B_{\beta s}$ for all the children segments, and the symbol counts $\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}$. The natural codes and the ordered pairs representing segment boundaries can be constructed using pointer manipulations, which use $O(|S|)$ words of memory for the entire recursion (with our computational model, a word has $k = O(\log(N))$ bits). The $MDL_{\beta s}$ and $\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}$ variables use $O(1)$ words of memory per stage in the recursion. Because the typical maximal depth of the prefix tree $T$ is $O(\log(N))$ (see Section 1.3.6), Phase I typically uses $O(\log(N))$ words of memory for the entire recursion. The worst-case recursion depth is $O(N)$, using $O(N)$ words of memory for the entire recursion.

Phase II needs to store the array $U$, along with symbol counts $\{n_s^\alpha(x)\}_{\alpha \in \mathcal{X}}$, $\forall s \in S$ for the sequential KT estimator [44]. With our computational model, $U$ uses $O(N)$ words.
of memory, and the symbol counts use $O(|\hat{S}|)$ words of memory. Because $|\hat{S}| \leq O(N)$ (see the proof of Theorem 1), the memory use of Phase II is $O(N)$ words.

In practice, we usually have $|\hat{S}| \ll N$ and the maximal depth of $T$ is $O(\log(N))$. Therefore, most of the memory is used for storing $x$, $y$, $T$, $V$, and $U$. The recursion of Phase I and the symbol counts of Phase II usually use much less memory.

2.4 Properties

2.4.1 Redundancy

The ML entropy of the input sequence $x$ w.r.t. a tree source structure $S$ is defined as

$$\hat{H}_S(x) \triangleq \sum_{s \in S} n_s(x) \hat{H}_s(x),$$

where $n_s(x)$ is the number of symbols in $x$ generated by state $s$, and $\hat{H}_s(x) = H(\{p_s^\alpha(x)\}_{\alpha \in X})$ as defined in (2.3). The redundancy of our unbounded variant over the ML entropy of $x$ w.r.t. the MDL structure $\hat{S}$ is upper bounded by $2|\hat{S}| - 1$ bits of model redundancy for representing $\hat{S}$, $\frac{1}{2} \log(N) + O(1)$ bits of parameter redundancy for representing each parameter (see Lemma 1), up to $|\hat{S}| \log(|X|)$ bits for directly encoding symbols with unknown contexts, and $O(1)$ bits of expected coding redundancy [10]. Therefore, the unbounded variant achieves the pointwise redundancy bound (1.2). This result was proved for CTW-MDL [14, Theorem 1], and also applies to Nohre’s method [5].

**Theorem 5 ([14])** Neglecting coding redundancy, the pointwise redundancy $\rho(x)$ of the unbounded variant over the ML entropy of the input sequence $x$ w.r.t. the MDL tree source structure $\hat{S}$ satisfies

$$\rho(x) \leq |\hat{S}| \left| \frac{|X| - 1}{2} \log(N) + O(1) \right|. \quad (2.11)$$

Because $\hat{S}$ is a globally optimal MDL tree source structure, the pointwise redundancy w.r.t. any other tree source satisfies (2.11). In particular, the pointwise redundancy over
the ML entropy of \( x \) w.r.t. the actual tree source structure \( S \) satisfies (2.11). Therefore, the unbounded variant also achieves the expected redundancy bound (1.1).

### 2.4.2 Optimality and statistical inference

Willems et al. [14] (see also Aberg [15]) proved that, under mild conditions, the MDL structure \( \hat{S} \) estimated by CTW-MDL [9, 14] asymptotically converges to the actual tree source structure \( S \) with probability 1. Our bounded variant considers the same class of tree sources as CTW-MDL. However, CTW-MDL assumes that the depth-\( D_{\text{max}} \) contexts of all symbols in \( x \) are available, while our unbounded variant does not assume knowledge of \( x_{1-D_{\text{max}}}^{0} \). Despite this difference between the two algorithms, if the actual tree source \( \{S, \Theta\} \) is irreducible, then the subsequence \( x_{1}^{D_{\text{max}}} \) is a recurrent state (a typical subsequence [1]). Asymptotically, all typical length-\( D_{\text{max}} \) subsequences will occur \( O(N) \) times in \( x \), and since the bounded variant of BWT-MDL only examines segments whose depth is \( D_{\text{max}} \) or less, the differences in ML conditional probabilities occurring in segments examined by the two algorithms will asymptotically converge to zero; hence the same tree source structure \( \hat{S} \) will be estimated. Therefore, under mild conditions, the MDL structure \( \hat{S} \) estimated by our bounded variant asymptotically converges to the actual tree source structure \( S \) with probability 1.

BWT-MDL computes the MDL tree source structure in \( O(N) \) complexity. MDL enjoys various optimal asymptotic properties [50]; hence, our method can be used for universal estimation, classification, and other problems in statistical inference. In particular, because our method is an asymptotically optimal estimator of tree source structure and parameters, BWT-MDL can thus be interpreted as optimal for universal estimation, prediction, classification, and modeling [2, 50]. Possible specific applications include speech and image recognition, and genome analysis.
2.4.3 Comparison to other source coding methods

We summarized the attributes of BWT-MDL and several other source coding methods over a binary alphabet in Table 1.1. The redundancies are similar, except that they are relative to the maximal context depth that is supported. Since a bounded context depth algorithm, i.e., $D_{\text{max}}(N) < N$, is not truly universal, it is most fair to compare our unbounded variant to other unbounded methods [6,8,11,12]. BWT-MDL has lower computational complexity, in particular worst-case complexity, because the BWT is $O(N)$, and Phase I is also $O(N)$ when the MDL routine scans the (compact) prefix tree. The unfavorable attributes of our algorithm are that its compression quality is slightly worse than algorithms that use the mixture approach (Section 1.4.3), and that it is nonsequential.

2.4.4 Decoder memory versus compression tradeoff

In Section 2.2.3, the goal was to reduce the coding length as much as possible. In this section, we assume that the decoder uses $O(|\hat{S}|)$ memory,\(^2\) and we are willing to give up some of the compression in order to gain a substantially more memory-efficient decoder (our encoder computes the BWT, so its memory complexity is $O(N)$ regardless of such tradeoffs).

Volf and Willems [9] proposed to optimize the tradeoff between compression and decoder memory by imposing a constraint $C$ on the number of leaves in the tree source structure estimated in CTW-MDL. They do so by scanning the tree in a depth-first search, and maintaining for every state a list containing for $c = 1, \ldots, C$ the minimal coding length for a subtree with $c$ leaves. After finding the number of leaves in the

\(^2\)A trivial algorithm can implement the decoder using $O(|\hat{S}|)$ memory and $O(N \log(N))$ complexity for typical input sequences.
subtrees rooted at $\lambda$, they recursively run their method on each of those subtrees. This recursive so called *yo-yo method* has superlinear complexity in the encoder.

We propose to weigh-in the use of memory in the decoder. Increasing the weight on state-splitting in the MDL structure beyond the associated coding cost will produce tree sources with fewer states. Consider our unbounded variant with a binary alphabet. In Section 2.2.3, $|\text{natural}_S| - \log(p_S(x))$ was the target function of the MDL minimization (2.1). Instead, we define our target function as

$$
\psi_S(x) \triangleq -\gamma|\mathcal{S}| + |\text{natural}_S| - \log(p_S(x)),
$$

(2.12)

where $\gamma$ is some positive constant (for $\gamma = 0$, this cost coincides with the MDL cost (2.1)). As specified above, we assume that the decoder uses $O(|\hat{S}|)$ memory. Although the $\gamma|\mathcal{S}|$ term in (2.12) does not accurately represent the amount of memory used by the decoder, it is a useful approximation.

Instead of paying 1 bit per decision (split or merge), we run Phase I of the basic algorithm with (2.6) modified to

$$
\overline{MDL}_s = \begin{cases} 
\frac{2}{2} + 1 + KT_s & \text{if } |\text{segment}_s| = 1 \\
\frac{2 + 2}{2} + \min \left\{ 1_{\{x_1 = s\}} + MDL_{0s} + MDL_{1s}, \quad KT_s \right\} & \text{otherwise}
\end{cases}
$$

(2.13)

We then proceed to Phase II and use the natural code as usual (the different weighting of splits and merges is only considered during the minimization). The following result holds.

**Theorem 6** For a binary alphabet, the modified form of Phase I that uses (2.13) yields a tree source structure that optimizes $\psi$.

*Proof:* For the unbounded variant with a binary alphabet, the total number of nodes is $2|\hat{S}| - 1$. Therefore, the additional $\frac{2}{2}$ cost is paid $2|\hat{S}| - 1$ times in (2.13), for a total
of \(\frac{2}{1}(2|\hat{S}| - 1)\) bits. This coincides with the first term of (2.12), up to a \(O(1)\) constant. Lastly, because the outcome of the modified Phase I is obtained by applying dynamic programming, the result of the minimization is a global minimum for \(\psi_S(x)\). □

**Remark 3** We can approximate the coding length \(\psi_S(x)\) by using different weightings for leaves and internal nodes in the explicit description of \(S\), instead of the natural code [5, 13], which encodes each node with 1 bit. This approach is similar to using different weightings in CTW [13, Section VII].

### 2.5 Conclusions and Future Work

Instead of paying the penalty for transitions between segments of the BWT output, we used the BWT as an efficient computational tool for pruning context trees. We described an \(O(N)\) complexity algorithm for pruning an unbounded depth context tree, using efficient prefix tree algorithms. Furthermore, in contrast to previous algorithms that required \(O(N \log(N))\) typical (and often \(O(N^2)\) worst-case) complexity for determining which state generated each of the symbols of an input sequence \(x\), we overcame that problem by exploiting the segment structure of the BWT output \(y\), and its relation to \(x\). The result of these improvements is an \(O(N)\) encoder for the semipredictive approach [5, 7, 9, 14]. Our BWT-MDL method can also be used to provide a tradeoff between memory use in the decoder and compression quality.

BWT-MDL opens up a range of opportunities for future research. First, practical aspects need to be considered. While the theoretical results predict competitive compression performance on large files, achieving similar results for shorter files would require studying the use of various heuristics for practical sources. Second, the method described in this chapter is nonsequential and requires preprocessing a length-\(N\) block with the BWT. We hope that our \(O(N)\) nonsequential semipredictive algorithm will motivate
work on $O(N)$ algorithms for the “plug-in” and mixture approaches. Third, BWT-MDL should be further evaluated for universal estimation, prediction, classification, and other problems in statistical inference. Lastly, we leave the development of low complexity decoders as an important open problem.
CHAPTER 3

TWO-PART CODES WITH LOW WORST-CASE REDUNDANCIES FOR DISTRIBUTED COMPRESSION OF I.I.D. SEQUENCES

3.1 Introduction

Source coding algorithms often model the input as an independent and identically distributed (i.i.d.) sequence. Therefore, universal compression of i.i.d. sequences with unknown parameters has received much research attention [28, 44, 51–53]. Rissanen [28] proved that, for length-N sequences, the worst-case redundancy (WCR) of any lossless source code over the maximum likelihood coding length is $\frac{1}{2} \log(N) + O(1)$ bits per unknown parameter, except for a set of parameters whose volume vanishes as $N \to \infty$. Rissanen also mentioned that this bound is asymptotically achieved by Krichevsky-Trofimov (KT) probability estimates [44]. With the KT method, the probability of each symbol of the sequence is estimated sequentially, taking into account the number of times the different symbols have appeared before the current symbol. The asymptotic optimality of KT estimates, along with their sequentiality, makes them a natural candidate whenever i.i.d. sequences need to be compressed universally.

Unfortunately, in some applications it is difficult to maintain the symbol counts sequentially. Consider a distributed compression system with several encoders, each pro-
cessing a portion of an i.i.d. sequence. While the overall compression quality is the main concern in this system, communication among encoders is also costly. Therefore, it is infeasible for the encoders to sequentially track the symbol counts together while they process the sequence.

An alternative to KT estimates are two-part codes [28, 51–53], which first encode a quantized estimate of the parameters and then encode the sequence based on the quantized parameter estimate. Two-part codes can be implemented in a distributed compression system because there is no need to maintain the symbol counts sequentially. Instead, each encoder preprocesses its portion of the sequence and sends the total symbol counts for its portion to the other encoders. The parameter estimate (and the first part of the code) is then based on information gathered by all the encoders, taking the entire sequence into account. The second part of the code is implemented independently by each of the encoders. Therefore, in distributed compression systems, two-part codes have a significant advantage over KT estimates.

In this chapter, we study two-part codes for i.i.d. sequences with unknown parameters. We concentrate on quantized-parameter (QP) two-part codes, which have a particularly simple structure, and show how these two-part codes can be used for universal distributed lossless source coding of i.i.d. sequences. While the WCR that we achieve is not as good as Rissanen’s bound, the loss turns out to be minor.

In the remainder of the chapter, we mainly concentrate on Bernoulli sequences (over binary alphabets). In Section 3.2, we define our notation and describe Rissanen’s bound.\footnote{The bound that will be described is more precise than the bound (1.1), which was provided in Section 1.2.} Section 3.3 defines QP codes and explains how they can be used for distributed lossless source coding. In Section 3.4 we discuss a QP code for i.i.d. Bernoulli sequences that quantizes the parameter estimate according to Jeffreys’ prior and prove that its WCR
is asymptotically upper bounded by 1.221 bits above Rissanen’s bound. In Section 3.5, we prove that the best WCR attainable among QP codes is 1.047 bits above the bound. Section 3.6 generalizes our results to finite alphabets, and Section 3.7 discusses our results and presents some problems for future work.

### 3.2 Bounds on Universal Coding of Bernoulli Sequences

Consider a length-$N$ i.i.d. Bernoulli sequence $x = x_1 x_2 \cdots x_N$ over a binary alphabet $\mathcal{X} = \{0, 1\}$, i.e., $x_i \in \mathcal{X}$. We denote the number of times 0 and 1 appear in the sequence $x$ as $n^0(x)$ and $n^1(x)$, so $n^0(x) \triangleq \sum_{i=1}^{N} 1_{\{x_i=0\}}$ and $n^1(x) \triangleq \sum_{i=1}^{N} 1_{\{x_i=1\}} = N - n^0(x)$, where $1_{\{\}}$ denotes an indicator function. If the Bernoulli parameter is $\theta$, the probability for obtaining $x$ is

$$p(x; \theta) = \theta^{n^1(x)}(1 - \theta)^{n^0(x)}.$$  \hspace{1cm} (3.1)

Alternatively, $x$ can be assigned probabilities according to any probability mass function (PMF) $q(x)$ such that $q(x) \geq 0$, $\forall x \in \mathcal{X}^N$ and $\sum_{x \in \mathcal{X}^N} q(x) \leq 1$. We consider lossless compression (coding) of Bernoulli sequences for some fixed $N$.

Since the appearance of arithmetic coding [1, 10, 41], it has been common in the source coding community to separate modeling, which consists of assigning probabilities to sequences, and coding, which translates probability assignments to a uniquely decodable code [1]. Because the redundancy of arithmetic coding is essentially $O(1)$ bits [1, 10], in this chapter we neglect the redundancy of arithmetic coding\(^2\) and assume that $x$ is encoded with $l_q(x) = -\log(q(x))$ bits (even if $l_q(x) \notin \mathbb{N}$). We call $l_q(x)$ the **coding length**

\(^{2}\)In this chapter, we compare two-part codes all of whose WCRs are approximately 1 bit per parameter above Rissanen’s bound [28]. Full-scale compression systems usually process non-i.i.d. sources; a tree source may have unknown parameters for each of the states, resulting in a WCR proportional to the number of states. On the other hand, the multiparameter probability assignment for $x$ is fed into a single arithmetic encoder, incurring arithmetic coding redundancy only once. Therefore, the redundancy of arithmetic coding is negligible compared to the overall redundancy above Rissanen’s bound.
for the sequence $x$, and view $q(x)$ and $l_q(x)$ as corresponding to each other. Because we
neglect the redundancy of arithmetic coding, in this chapter we only deal with probability assignments. Accordingly, we use the term code to refer to a method for probability assignment.

When $\theta$ is known, we can use the code (probability assignment) $p(x; \theta)$ of (3.1). Using our assumption that the probability assignment $q(x)$ is encoded with $l_q(x) = -\log(q(x))$ bits, encoding $x$ requires $-\log(p(x; \theta))$ bits. In this case, the per-symbol expected coding length becomes the per-symbol binary entropy [1], which is

$$H(\theta) \triangleq -\theta \log(\theta) - (1 - \theta) \log(1 - \theta)$$

bits per symbol.

When $\theta$ is unknown, the code should be universal over all possible parameters. Using (3.1), different parameter values lead to different probabilities for $x$, hence different coding lengths. The minimum coding length is obtained by using the maximum likelihood (ML) parameter,

$$\theta_{ML}(x) = \frac{n^1(x)}{N}.$$

We define the ML probability $p_{ML}(\cdot)$ as the PMF determined by the ML parameters. For Bernoulli sequences,

$$p_{ML}(x) \triangleq p(x; \theta_{ML}(x)).$$

Similarly, we define the ML coding length as

$$l_{ML}(x) \triangleq -\log(p_{ML}(x)) = NH(\theta_{ML}(x)).$$

However, $p(x; \theta_{ML}(x)) > p(x; \theta)$ for any $x$ and any $\theta \neq \theta_{ML}(x)$. Furthermore, because

$$\sum_{x \in \mathcal{X}^N} p(x; \theta) = 1$$

for any $\theta$,

$$\sum_{x \in \mathcal{X}^N} p(x; \theta_{ML}(x)) > 1,$$
which violates the Kraft inequality [1]. We conclude that, while the ML coding length is a useful benchmark for evaluating the performance of a universal code, it does not correspond to a valid probability assignment.

For any probability assignment \( q(x) \), the excess of the coding length \( l_q(x) \) above the ML coding length \( l_{ML}(x) \) is known as redundancy [28, 51, 52], defined as

\[
R_q(x) \triangleq l_q(x) - l_{ML}(x) = \log \left( \frac{p_{ML}(x)}{q(x)} \right).
\]  

(3.2)

The worst-case redundancy (WCR) for a probability assignment \( q(x) \) over all possible inputs is

\[
\max_{x \in \mathcal{X}^N} R_q(x) = \max_{x \in \mathcal{X}^N} \log \left( \frac{p_{ML}(x)}{q(x)} \right).
\]

The WCR is minimized by the normalized maximum likelihood (NML) PMF \( q^*(\cdot) \) [51],

\[
q^*(x) = \frac{p_{ML}(x)}{\sum_{y \in \mathcal{X}^N} p_{ML}(y)}.
\]  

(3.3)

With the corresponding coding length \( l_{NML}(x) \triangleq -\log(q^*(x)) \), all inputs have the same redundancy,

\[
R^{\mathcal{X}}(N) = \log \left( \sum_{y \in \mathcal{X}^N} p_{ML}(y) \right),
\]  

(3.4)

which serves as a lower bound on the WCR.

The per-symbol Fisher information [1, 28, 51] is defined as

\[
I(\theta) \triangleq -\frac{1}{N} E \left[ \frac{\partial^2}{\partial \theta^2} \ln(p(x; \theta)) \right].
\]

For Bernoulli sequences, the Fisher information is [28]

\[
I(\theta) = \frac{1}{\theta(1 - \theta)}.
\]  

(3.5)

Rissanen [28] proved that for a general finite alphabet \( \mathcal{X} \)

\[
R^{\mathcal{X}}(N) = \frac{|\mathcal{X}|-1}{2} \log \left( \frac{N}{2\pi} \right) + \log \left( \int_{\Omega} \sqrt{I(\theta)} \, d\theta \right) + o(1),
\]  

(3.6)
where $\Omega$ is the set of possible parameter values, and $I(\cdot)$ is bounded in $\Omega$. For $|\mathcal{X}| = 2$, and taking $\Omega \subset (0,1)$ with bounded $I(\cdot)$ such that the integral is arbitrarily close to $\pi$, this reduces to

$$R^2(N) = \frac{1}{2} \log \left( \frac{\pi}{2} N \right) + o(1). \quad (3.7)$$

### 3.3 Quantized-Parameter Two-Part Codes

Rissanen [28] constructed a two-part code that asymptotically approaches the lower bound on the WCR (3.6) for parameters in the interior of $\Omega$. His construction is based on uniform quantization of $\theta_{ML}(x)$. Unfortunately, Rissanen’s construction has higher redundancies near the boundaries of $\Omega$; hence its WCR is much higher than the lower bound. In practice, parameters on the edges of the probability simplex are common (for example, in English text ‘q’ is almost always followed by ‘u’), so low redundancies for parameters near the boundaries of $\Omega$ have practical importance, in addition to being of theoretic interest.

Another approach for low WCRs is offered by the normalized maximum likelihood (NML) PMF $q^*(\cdot)$ (3.3). However, we are not aware of previous work on using NML in a distributed source coding system. Instead, in the remainder of the chapter we consider QP two-part codes, which are simple enough to be used for distributed compression yet flexible enough to provide low WCRs, including low redundancies for parameters near the boundaries of the parameter space $\Omega$. We introduce QP codes in Section 3.3.1 and explain how to adapt these codes to distributed source coding systems in Section 3.3.2. The divergence and redundancy properties of QP codes are discussed in Section 3.3.3.

#### 3.3.1 Structure of quantized-parameter (QP) two-part codes

In the first part of the code, we describe $\theta_{ML}(x)$ as a quantized value $\hat{\theta}_{ML}(x)$. We use a nonuniform quantizer with $K$ bins $[b_0, b_1), [b_1, b_2), \ldots, [b_{K-1}, b_K]$, where $b_0 = 0$
and $b_K = 1$, and $K$ representation levels $r_k \in [b_{k-1}, b_k]$, $k \in \{1, \ldots, K\}$. We denote this quantizer by $Q_K = \{\{b_k\}_{k=0}^K, \{r_k\}_{k=1}^K\}$, and view it as a quantization operator; if $\theta_{ML}(x) \in [b_{k-1}, b_k)$, we define the outcome of the quantization as $Q_K(\theta_{ML}(x)) = \tilde{\theta}_{ML}(x) = r_k$. The first part of the code consists of $\log(K)$ bits describing the bin index $k$, which corresponds to assigning equal probabilities $\frac{1}{K}$ to all $K$ bins.

In the second part of the code, the input sequence $x$ is encoded according to $\tilde{\theta}_{ML}(x)$. Each symbol $x_i$, $i \in \{1, \ldots, N\}$ is assigned the probability $\tilde{\theta}_{ML}(x) = r_k$ or $1 - \tilde{\theta}_{ML}(x) = 1 - r_k$ according to whether $x_i$ is 1 or 0, respectively. Therefore, the probability assigned to $x$ in the second part of the code is

$$p(x; \tilde{\theta}_{ML}(x)) = p(x; r_k) = (r_k)^{n_1(x)}(1 - r_k)^{n_0(x)}.$$  

(3.8)

Note that, given a fixed $r_k$, $p(x; r_k)$ is a PMF. We call these codes quantized-parameter two-part codes (QP codes).

### 3.3.2 Using quantized-parameter codes for distributed source coding

Consider a distributed compression system with $B$ encoders that process $B$ blocks of a single i.i.d. sequence. We want to reconstruct the $B$ blocks at $B$ decoders, where communication between encoders and decoders is carried out over a common channel. There may also be internal communications among the $B$ encoders and among the $B$ decoders; if the interencoder and interdecoder distances are small, internal communications are local and may be less costly than communications over the common channel. In this system, the overall compression quality, which determines the amount of communications over the common encoder-to-decoder channel, is the main concern. However, reducing internal communications is also important. We will show that QP codes can be implemented on such a system with minor internal communications.
Denote a block by \( x^j_i = x_i x_{i+1} \ldots x_j \), where \( j \geq i \). Denote the \( B \) disjoint blocks by \( x_{i_0+1}^{i_1}, x_{i_1+1}^{i_2}, \ldots, x_{i_{B-1}+1}^{i_B} \), where \( i_0 = 0 \leq i_1 \leq \ldots \leq i_B = N \). The values of \( n^0(x) \) and \( n^1(x) \) can be computed by \( n^0(x) = \sum_{j=1}^{B} n^0(x_{i_{j-1}+1}^{i_j}) \) and \( n^1(x) = \sum_{j=1}^{B} n^1(x_{i_{j-1}+1}^{i_j}) \).

Given \( n^0(x) \) and \( n^1(x) \), we can compute \( \theta_{ML}(x) \).

We now describe a distributed QP code for a system with \( B \) encoders. The \( B \) encoders share \( \{n^0(x_{i_{j-1}+1}^{i_j})\}_{j=1}^{B} \) and \( \{n^1(x_{i_{j-1}+1}^{i_j})\}_{j=1}^{B} \) with each other, which enables each encoder to compute \( \theta_{ML}(x) \). All \( B \) encoders then use a common quantizer \( Q_K \) to determine \( \tilde{\theta}_{ML}(x) = Q_K(\theta_{ML}(x)) \). In the first part of the distributed QP code, \( \log(K) \) bits are used to encode the bin index \( k \). This may be done by a designated encoder, e.g., the one with \( j = 1 \). In the second part of the distributed QP code, each encoder \( j \) assigns probabilities to the symbols of \( x_{i_{j-1}+1}^{i_j} \) and feeds them to an arithmetic encoder, i.e., there are \( B \) arithmetic encoders. The probability assigned by encoder \( j \) to the symbols in \( x_{i_{j-1}+1}^{i_j} \) is

\[
p(x_{i_{j-1}+1}^{i_j}; \tilde{\theta}_{ML}(x)) = \left( \tilde{\theta}_{ML}(x) \right)^{n^1(x_{i_{j-1}+1}^{i_j})} \left( 1 - \tilde{\theta}_{ML}(x) \right)^{n^0(x_{i_{j-1}+1}^{i_j})}.
\]

The product of the probabilities assigned by all \( B \) encoders is thus

\[
\prod_{j=1}^{B} p(x_{i_{j-1}+1}^{i_j}; \tilde{\theta}_{ML}(x)) = \left( \tilde{\theta}_{ML}(x) \right)^{n^1(x)} \left( 1 - \tilde{\theta}_{ML}(x) \right)^{n^0(x)}.
\]

Because this result is identical to the probability assigned in the second part of a nondistributed QP code (3.8), using a single quantized ML parameter \( \tilde{\theta}_{ML}(x) \) for all \( B \) encoders enables us to obtain the same redundancy for distributed QP codes as we would obtain with nondistributed QP codes. Furthermore, we can also support distributed QP decoding; by sharing \( \tilde{\theta}_{ML}(x) \) among all \( B \) decoders, each decoder \( j \) can reconstruct \( x_{i_{j-1}+1}^{i_j} \). Combining all \( B \) reconstructed blocks provides \( x \).

With distributed QP codes, the \( B \) encoders need only communicate \( \{n^0(x_{i_{j-1}+1}^{i_j})\}_{j=1}^{B} \) and \( \{n^1(x_{i_{j-1}+1}^{i_j})\}_{j=1}^{B} \) to each other; hence there is \( O(B \log(N/B)) \) internal communications among the encoders. The \( B \) decoders need only share the bin index \( k \), which
requires $O(\log(K))$ internal communications. We later show that $K = O(\sqrt{N})$, so the internal communications required among components of such a distributed source coding system is minor. In contrast, KT estimates [44] require $O(N)$ internal communications, which may be unreasonable in some scenarios.

### 3.3.3 Divergence and redundancy of quantized-parameter codes

Define the Kullback-Leibler divergence [1] between two Bernoulli PMFs with parameters $p$ and $q$ as

$$D(p\|q) \triangleq p \log \left( \frac{p}{q} \right) + (1 - p) \log \left( \frac{1 - p}{1 - q} \right). \quad (3.9)$$

Denote the coding length and the redundancy for $x$ of the two-part code based on $Q_K$ by $l_{Q_K}(x)$ and $R_{Q_K}(x)$, respectively. The following expressions for $l_{Q_K}(x)$ and $R_{Q_K}(x)$ follow immediately from the foregoing definitions.

**Lemma 2** For a QP code based on $Q_K$,

$$l_{Q_K}(x) = \log(K) - n^1(x) \log(\bar{\theta}_{ML}(x)) - n^0(x) \log(1 - \bar{\theta}_{ML}(x))$$

and

$$R_{Q_K}(x) \triangleq l_{Q_K}(x) - l_{ML}(x) = \log(K) + N D(\theta_{ML}(x)\|\bar{\theta}_{ML}(x)). \quad (3.10)$$

**Remark 4** A similar formulation was provided by Chou et al. [52].

It follows from Lemma 2 that, for a fixed quantizer $Q_K$, the WCR occurs for $x$ that maximizes $D(\theta_{ML}(x)\|\bar{\theta}_{ML}(x))$. The WCR is therefore determined by the distortion introduced by the quantizer, where the distortion measure is the divergence $D(\theta\|Q_K(\theta))$ between the input $\theta$ and the quantized output $Q_K(\theta)$ [52].

**Worst-case divergence and redundancy:** We define the **worst-case bin divergence** (WCBD) of quantizer bin $k$ as

$$D_{Q_K}(k) \triangleq \max_{\theta \in [b_{k-1}, b_k)} D(\theta\|Q_K(\theta)) = \max \{ D(b_{k-1}\|r_k), D(b_k\|r_k) \},$$
where the equality follows from the monotonicity of the divergence [1]. The \textit{worst-case divergence} (WCD) for quantizer $Q_K$ is defined as

$$D_{Q_K} \triangleq \max_{\theta \in [0, 1]} D(\theta \| Q_K(\theta)) = \max_{k \in \{1, \ldots, K\}} \{D_{Q_K}(k)\}. \quad (3.11)$$

In accordance with (3.10),

$$R_{Q_K}(x) \leq \log(K) + N \cdot D_{Q_K}. \quad (3.12)$$

Accordingly, for length-$N$ inputs we define an \textit{upper bound on the worst-case redundancy} (WCR) as

$$R_{Q_K}(N) \triangleq \log(K) + N \cdot D_{Q_K}. \quad (3.13)$$

\textbf{Operational divergence and redundancy:} Whereas $\theta_{ML}(x) = n^1(x)/N$, bin edges need not equal multiples of $1/N$. Because $\theta_{ML}(x)$ may not “land” on a bin edge, the upper bound in (3.12) may not be achieved. We therefore introduce operational versions of the WCBD, WCD, and WCR that account for these effects.

The worst-case redundancy (in an operational sense) occurs for $x \in \mathcal{X}^N$ that maximizes $D(\theta_{ML}(x)\|\tilde{\theta}_{ML}(x))$. Let $\xi(k, N) \triangleq \{x \in \mathcal{X}^N : \theta_{ML}(x) \in [b_{k-1}, b_k)\} = \{x \in \mathcal{X}^N : Q_K(\theta_{ML}(x)) = r_k\}$. We denote the \textit{operational worst-case bin divergence} (WCBD$^O$) for inputs in $\xi(k, N)$ by

$$D_{Q_K}^O(k, N) \triangleq \max_{x \in \xi(k, N)} D(\theta_{ML}(x)\|r_k),$$

and by $D_{Q_K}^O(N)$ the \textit{operational worst-case divergence} (WCD$^O$) for length-$N$ inputs of a code based on $Q_K$, i.e.,

$$D_{Q_K}^O(N) \triangleq \max_{x \in \mathcal{X}^N} D(\theta_{ML}(x)\|\tilde{\theta}_{ML}(x)) = \max_{k \in \{1, \ldots, K\}} D_{Q_K}(k, N).$$

Using Lemma 2, the corresponding WCR for length-$N$ inputs of a code based on $Q_K$ is

$$R_{Q_K}^O(N) \triangleq \max_{x \in \mathcal{X}^N} R_{Q_K}(x) = \log(K) + N \cdot D_{Q_K}^O(N).$$

60
The WCBD and WCBD\(^O\) can be directly related at the bin level. Let \(\lceil \cdot \rceil\) and \(\lfloor \cdot \rfloor\) denote rounding up and down, respectively. Because \(\theta_{ML}(x) = \lfloor [N b_{k-1}] / N, [N b_k] / N \rfloor \subseteq [b_{k-1}, b_k)\) is represented by \(r_k\), we have

\[
D_{Q_K}^O(k, N) = \max_{N b_{k-1} \leq n \leq N b_k, n \in \mathbb{N}} D(n / N \| r_k) \\
= \max \{ D([N b_{k-1}] / N \| r_k), D([N b_k] / N \| r_k) \} \\
\leq \max \{ D(b_{k-1} \| r_k), D(b_k \| r_k) \} \\
= D_{Q_K}(k).
\]

(3.14)

We summarize the relations between operational and nonoperational quantities in the following lemma.

**Lemma 3** For a QP code based on \(Q_K\),

\[
D_{Q_K}^O(k, N) \leq D_{Q_K}(k),
\]

(3.15)

\[
D_{Q_K}^O(N) \leq D_{Q_K},
\]

(3.16)

and

\[
R_{Q_K}^O(N) \leq R_{Q_K}(N).
\]

(3.17)

**Proof:** The first result (3.15) follows from (3.14). The remaining results (3.16) and (3.17) follow from (3.15) and the foregoing definitions. \(\square\)

**Remark 5** The main results of this chapter are obtained using the intuition that in “large” bins, i.e., for \(N(b_k - b_{k-1}) \gg 1\), the WCBD and WCBD\(^O\) are similar. We bound the WCBDs, hence the WCD and WCR, which produces the corresponding result for the WCR.

Among codes with \(K\) bins, the optimum quantizer with \(K\) bins \(Q_K^*\) minimizing \(R_{Q_K}(N)\) minimizes the WCD of the second part of the code, i.e., \(Q_K^* = \arg\min_{Q_K} \{ D_{Q_K} \} \).
The corresponding optimal WCD and WCR are

\[ D_K = D_{Q_K} = \min_{Q_K} \{ D_{Q_K} \} \]  \hspace{1cm} (3.18)

and

\[ R_K(N) = \log(K) + N \cdot D_K, \]  \hspace{1cm} (3.19)

respectively. We choose \( K \) to minimize \( R_K(N) \).

A quantizer \( Q_K \) is good if all worst-case bin divergences (WCBDs) of all the bins are similar (see Section 3.5.1), i.e., \( D_{Q_K}(1) \approx D_{Q_K}(2) \cdots \approx D_{Q_K}(K) \); otherwise “good” bins are offset by “bad” bins, which raise the WCR. With QP codes, once we specify \( Q_K \), the coding length, the WCD, and the WCR follow. Although the class of QP codes is suboptimal, we later analyze the WCR and WCR of its codes and show that the loss incurred is minor.

**Remark 6** The optimal quantizer \( Q^*_K \) admits the following geometric interpretation. Consider the plot of the binary entropy in Figure 3.1. Let \( l_q(\theta) \) be the function defined by the straight line tangent to \( H(\theta) \) at \( \theta = q \). Then \( l_q(\theta) = (\theta - q) \log(\frac{1-q}{q}) + H(q) \), and it is easily verified that \( D(p\|q) = l_q(p) - H(p) \). We associate with quantizer \( Q_K \) a \( K \)-segment poly-line fit to the entropy function, with the \( k \)-th segment tangent to \( H(\theta) \) at \( \theta = r_k \), and its end points at \( \theta = b_{k-1} \) and \( \theta = b_k \). Using (3.11), \( D_{Q_K} \) is the \( L_\infty \) distance of the poly-line fit from the entropy function. In particular, \( Q^*_K \) corresponds to the \( K \)-segment poly-line fit whose \( L_\infty \) distance from \( H(\theta) \) is minimum.

### 3.4 Code Motivated by Jeffreys’ Prior

The QP code described in this section is based on Stine and Foster’s work [53, Section 3]. Instead of providing the best quantizer for the parameter estimate, as will be done in Section 3.5, we provide a practical quantization scheme, motivated by a property of
Figure 3.1 A geometric interpretation of bin divergences.

Jeffreys’ prior [51], which roughly equalizes the WCBDs of all the bins. We will show that this scheme is simple to implement and that its \( \overline{\text{WCR}} \) and WCR are only slightly greater than Rissanen’s bound (3.7).

Jeffreys’ prior [28, 51, 53] over a parameter set \( \theta \in \Omega \) associated with a PMF \( p(x; \theta) \) is defined as

\[
\pi(\theta) \triangleq \frac{\sqrt{|I(\theta)|}}{\int_{\eta \in \Omega} \sqrt{|I(\eta)|}} d\eta,
\]

where \( I(\theta) \) is the Fisher information for \( p(x; \theta) \). Using the Fisher information previously defined for Bernoulli sequences (3.5), Jeffreys’ prior has the useful property that if two
intervals $[a, b], [c, d] \subseteq [0, 1]$ satisfy
\[
\int_a^b \pi(\theta) \, d\theta = \int_c^d \pi(\theta) \, d\theta,
\]
then $D(a\|b) \approx D(c\|d)$ [51]. Define the mapping
\[
\eta(\theta) \triangleq \frac{1}{\pi} \int_0^\theta \sqrt{I(u)} \, du. \tag{3.21}
\]
It follows that for $\theta_a$, $\theta_b$, $\theta_c$, and $\theta_d$ satisfying $\eta(\theta_b) - \eta(\theta_a) = \eta(\theta_d) - \eta(\theta_c)$, we have
$D(\theta_a\|\theta_b) \approx D(\theta_c\|\theta_d)$. This observation motivates the following quantization scheme for $\theta_{ML}(x)$. We define $\eta_{ML}(x) \triangleq \eta(\theta_{ML}(x))$, uniformly quantize $\eta_{ML}(x)$ to obtain $\bar{\eta}_{ML}(x)$, and compute the representation level $\bar{\theta}_{ML}(x) = \theta(\bar{\eta}_{ML}(x))$, where $\theta(\eta)$ is the inverse function of $\eta(\theta)$. Uniform quantization in the $\eta$ bins implies similar WCBDs in the $\theta$ bins, hence the WCD and WCR will be small.

**Remark 7** It can be shown that
\[
D(p\|q) = \int_q^p (p - u) \frac{I(u)}{\ln(2)} \, du,
\]
where $p$ is a bin edge and $q$ is a representation level. If in the range $\theta \in (p, q)$ we have $I(\theta) \approx I^*$, then $D(p\|q) \approx (p - q)^2 \frac{I^*}{2 \ln(2)}$. In order to obtain equal divergences for all bins, i.e., $D(p\|q) = D^*$, we need $|p - q| \sqrt{I^*} \approx \sqrt{2D^* \ln(2)}$. Using (3.20), we conclude that Jeffreys’ prior is based on a second-order approximation of the divergence. Alternatively, we can think of the Fisher information as the curvature (second derivative) of the entropy function; using the geometric interpretation of Remark 6, averaging over the square root of the curvature allows to approximately equalize the divergences.

The remainder of the section is organized as follows. In Section 3.4.1 we compute the quantization bins and representation levels explicitly. For a fixed $K$, the WCD of this quantizer is bounded in Section 3.4.2. In Section 3.4.3 we optimize the number of bins, and Section 3.4.4 discusses operational aspects.
3.4.1 Quantization bins and representation levels

Following Stine and Foster [53], the integration over \( \sqrt{I(u)} \) in the Bernoulli case (3.5) is

\[
\int_0^\theta \sqrt{I(u)} \, du = \int_0^\theta \frac{1}{\sqrt{u(1-u)}} \, du = 2 \arcsin\left(\sqrt{\theta}\right).
\]

Using (3.21), we have the mappings \( \eta(\theta) = \frac{2}{\pi} \arcsin\left(\sqrt{\theta}\right) \) and \( \theta(\eta) = \sin^2\left(\frac{\pi}{2} \eta\right) \). In order to uniformly quantize \( \eta_{ML}(x) \), we partition the bins as

\[
\left[0, \frac{1}{K}\right), \ldots, \left[\frac{k-1}{K}, \frac{k}{K}\right), \ldots, \left[\frac{K-1}{K}, 1\right).
\]

which corresponds to quantizing \( \theta_{ML}(x) \) with the bins

\[
\left[0, \sin^2\left(\frac{\pi}{2K}\right)\right), \ldots, \left[\sin^2\left(\frac{\pi}{2K}(k-1)\right), \sin^2\left(\frac{\pi}{2K}k\right)\right), \ldots, \left[\sin^2\left(\frac{\pi}{2K}(K-1)\right), 1\right).
\]

We place the representation level of each bin in the middle of the bin in the \( \eta(\cdot) \) domain.\(^3\)

Therefore,

\[
b_k = \sin^2\left(\frac{\pi k}{2K}\right)
\]

and

\[
r_k = \sin^2\left(\frac{\pi(2k-1)}{4K}\right).
\]

Note that, although \( k \) is an (integer) bin index, \( b_k \) and \( r_k \) are defined for a real-valued \( k \).

We define the Jeffreys quantizer for a given \( K \) according to (3.22) and (3.23), and denote this quantizer by \( \mathcal{J}_K \). Following our previous notation, we denote the WCD of \( \mathcal{J}_K \) by \( D_{\mathcal{J}_K} \) and the WCR for length-\( N \) inputs of a two-part code based on \( \mathcal{J}_K \) by \( R_{\mathcal{J}_K}(N) \). We emphasize that we only need to quantize \( \theta_{ML}(x) \) once for the entire length-\( N \) sequence.

3.4.2 Worst-case divergence

Instead of computing \( D(\theta_{ML}(x)\|\bar{\theta}_{ML}(x)) \) for every input \( x \in \mathcal{X}^N \), the following lemma enables us to focus on the maximum value of \( D(b_k\|r_k) \) for \( k \in \{1, \ldots, K\} \).

\(^3\)In Section 3.5.3, we will decrease the WCR slightly by modifying the representation level.
Lemma 4 For $\mathcal{J}_K$ we have

\[ D_{\mathcal{J}_K} = \max_{k \in \{1, \ldots, K\}} D(b_k \| r_k). \]  

(3.24)

Proof: With Jeffreys quantization, owing to symmetry we have $D(b_k \| r_k) = D(b_{K-k} \| r_{K-k+1})$, so

\[ \{D(b_{k-1} \| r_k)\}_{K}^{K-1} = \{D(b_0 \| r_1), \ldots, D(b_{K-1} \| r_K)\} = \{D(b_K \| r_K), \ldots, D(b_1 \| r_1)\} = \{D(b_k \| r_k)\}_{K}^{K-1}. \]  

(3.25)

Combining (3.11) with (3.25) yields (3.24). \(\square\)

Now that the structure of $\mathcal{J}_K$ has been determined, our goal is to bound $D_{\mathcal{J}_K}$. We begin with a lemma, which is proved in Appendix C.

Lemma 5 For the Jeffreys quantizer $\mathcal{J}_K$, $\frac{4}{dz}D(b_z \| r_z) \leq 0$ where $z \in \mathbb{R}$, $1 \leq z \leq K$.

Using Lemmas 4 and 5, we conclude that $D_{\mathcal{J}_K} = D(b_1 \| r_1)$. A detailed calculation in Appendix E then establishes the following explicit bound.

Theorem 7 The worst-case divergence of the Jeffreys quantizer $\mathcal{J}_K$ satisfies

\[ D_{\mathcal{J}_K} = \frac{\pi^2}{K^2} \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) + O \left( \frac{1}{K^4} \right). \]

3.4.3 Number of bins

Combining Lemma 2 and Theorem 7,

\[ R_{\mathcal{J}_K}(N) = \log(K) + \frac{\pi^2}{K^2} \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) N + O \left( \frac{N}{K^4} \right), \]

which is minimized by

\[ K_{p}^*(N) = \sqrt{2\pi^2 \ln(2) \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right)} N + o(1). \]  

(3.26)
Note that the optimal number of bins \( K^*_J(N) = O(\sqrt{N}) \). We thus define the Jeffreys code according to the bin edges in (3.22), the representation levels in (3.23), and \( \hat{K}_J(N) = [K^*_J(N)] \in \mathbb{N} \) bins. Denote the WCR of the Jeffreys code by \( R_J(N) \). Because the second derivative of \( R_{J_K}(N) \) w.r.t. \( K \) is \( O \left( \frac{1}{N} \right) \) for \( K = K^*_J(N) \),

\[
R_J(N) = R_{J_{K^*_J}(N)}(N) + O \left( \frac{1}{N} \right) = \frac{1}{2} \log \left[ 2\pi^2 e \ln(2) \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) N \right] + O \left( \frac{1}{N} \right).
\]

Comparing this to Rissanen’s bound (3.7), the additional redundancy that is introduced is

\[
R_J(N) - R^2(N) = \frac{1}{2} \log \left[ 2\pi^2 e \ln(2) \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) \right] - \frac{1}{2} \log \left( \frac{\pi}{2} \right) + O \left( \frac{1}{N} \right)
\]

\[
= 1.221 + O \left( \frac{1}{N} \right) \text{ bits.}
\]

**Theorem 8** The WCR of the Jeffreys code is \( 1.221 + O \left( \frac{1}{N} \right) \) bits above Rissanen’s bound.

### 3.4.4 Operational Jeffreys code

In Section 3.5, we will show that \( J_K \) is nearly optimal away from the boundaries of the parameter space; in the initial and end bins, \( J_K \) can be improved. However, the initial and end bins are small, i.e., few values of \( n^1(x) \) belong to each of those bins, so the operational divergences (WCBD\(^O\)s) of those bins may be substantially smaller than the WCBDs. Therefore, the WCBD\(^O\)s may be substantially smaller than the WCBDs precisely in the bins where improvement is most needed.

We evaluated the WCD\(^O\) and WCR of Jeffreys quantizers numerically. We checked a range of values for \( K \), and picked the optimum. We call a QP code based on \( J_K \), where \( K \) minimizes the WCR, an operational Jeffreys code; the WCR of operational Jeffreys codes appears in Figure 3.5 on page 79. It turns out that, for some values of \( N \) and \( K \), \( D_{J_K}^O(N) \) approaches the optimal WCD \( D_K \). Furthermore, the WCR of operational Jeffreys codes seems to approach the best attainable WCR and WCR among QP codes. The following conjecture is based on Section 3.5 and the intuition that was outlined before.
Conjecture 1 The WCR of operational Jeffreys codes is $1.047 + o(1)$ bits above Rissanen's bound.

Although the WCR of operational Jeffreys codes may seem promising, it has the substantial drawback that the optimal $K$ must be determined numerically. Numerical results suggest that using $K = \lceil \frac{5}{2} \sqrt{N} \rceil$ (see Section 3.5) usually provides a WCR that is near the optimum, but it is not clear whether this is always the case. Therefore, relying on the operational characteristics may not be the best way to go. Instead, Section 3.5 focuses on QP codes with low WCRs.

3.5 Optimal Two-Part Codes

In this section, we consider optimizing the quantization of the parameter estimate $\theta_{ML}(x)$. We begin by deriving necessary conditions for optimality in Section 3.5.1. Then, in Section 3.5.2, we prove that the best WCR and WCR attainable among QP codes are 1.047 bits above Rissanen's bound. Finally, in Section 3.5.3 we provide a slight modification of the Jeffreys code that asymptotically achieves the 1.047 lower bounds on the WCR and WCR.

3.5.1 Necessary conditions for optimality

Recall from (3.11), (3.18), and (3.19) that, given $K$, our objective is to determine $Q^*_K$, the optimal quantizer with $K$ bins. We derive necessary conditions for optimality of the bin edges and representation levels.

**Necessary condition on $r_k$:** Given the bin edges $\{b_k\}_{k=0}^K$, we optimize the representation levels $\{r_k\}_{k=1}^K$. Using (3.11), in the bin $[b_{k-1}, b_k)$

$$r_k^* = \arg\min_{r \in [b_{k-1}, b_k]} \max\{D(b_{k-1}||r), D(b_k||r)\},$$
but \( D(b_{k-1}||r) \) is monotone increasing in \( r \) and \( D(b_k||r) \) is monotone decreasing in \( r \) [1]. Therefore, at a minimum of \( \max\{D(b_{k-1}||r), D(b_k||r)\} \), the two divergence terms are equal, i.e., \( D(b_{k-1}||r_k^*) = D(b_k||r_k^*) \). Solving for \( r_k^* \) yields

\[
 r_k^* = \frac{1}{1 + 2 H(b_{k-1}||r_k) - H(b_{k-1}||r_k)}.
\]  \( (3.27) \)

**Necessary condition on \( b_k \):** Given the representation levels \( \{r_k\}_{k=1}^K \), we optimize the bin edges \( \{b_k\}_{k=0}^K \). Using (3.11), in an interval between two representation levels \([r_k, r_{k+1}]\)

\[
b_k^* = \arg \min_{b \in [r_k, r_{k+1}]} \max\{D(b||r_k), D(b||r_{k+1})\},
\]

but \( D(b||r_k) \) is monotone increasing in \( b \) and \( D(b||r_{k+1}) \) is monotone decreasing in \( b \) [1]. Therefore, at a minimum of \( \max\{D(b||r_k), D(b||r_{k+1})\} \), the two divergence terms are equal, i.e., \( D(b_k^*||r_k) = D(b_k^*||r_{k+1}) \). Solving for \( b_k^* \) yields

\[
b_k^* = \frac{\log \left( \frac{1-r_k}{1-r_{k+1}} \right)}{\log \left( \frac{r_{k+1}}{r_k} \frac{1-r_k}{1-r_{k+1}} \right)}.
\]  \( (3.28) \)

The following theorem provides necessary conditions for optimality. It can also be proved that these conditions are sufficient for optimality, and that there is a unique solution; therefore, any \( Q_K \) satisfying (3.27) and (3.28) is optimal.

**Theorem 9** Let \( Q_K^* \) be an optimal quantizer with \( K \) bins. Then (3.27) and (3.28) are satisfied.

**Remark 8** For operational optimality, we must modify the necessary conditions. However, the lower bound on the WCR of QP codes is asymptotically equal to the lower bound on the WCR (see Section 3.5.2). Therefore, we do not discuss operational optimality further.
Figure 3.2 The WCR above Rissanen’s bound of quantizer-optimal two-part codes approaches $\frac{1}{2} \log \left( \frac{3K}{2} \right) = 1.047$ bits.

Combining (3.27) and (3.28), we can obtain difference equations for the bin edges or representation levels. The conditions on $\{b_k\}_{k=0}^K$ and $\{r_k\}_{k=1}^K$ can also be solved with fixed-point iterations, yielding an algorithm similar to the Lloyd-Max quantization design algorithm [52, 54]. The method has two steps. In the first step, given $\{b_k\}_{k=0}^K$, we select optimum $\{r_k\}_{k=1}^K$ per (3.27). In the second step, given $\{r_k\}_{k=1}^K$, we select optimum $\{b_k\}_{k=0}^K$ per (3.28). We iterate over these two steps until the solutions satisfy some convergence condition. We also check a range of values for $K$, picking the optimum.

We call the two-part code based on the optimal quantizer that also uses the optimal number of bins the quantizer-optimal code. Figure 3.2 shows numerical results for the WCR of the quantizer-optimal code as a function of $N$. It appears that the WCR among QP codes approaches 1.047 bits above Rissanen’s bound. We now prove where this value comes from.
Figure 3.3 WCBDs for $K = \sqrt{N}$. Away from the boundaries of the parameter space $\Omega$, $D(b_{k-1}\|r_k)$ and $D(b_k\|r_k)$ are similar. There are large WCBDs near the boundaries of $\Omega$, but these regions become narrower as $N$ increases.

### 3.5.2 An attainable lower bound on the worst-case redundancy

**Theorem 10** The lowest $\mathrm{WCR}$ of any QP code is $1.047 + O(N^{-0.25})$ bits above Rissanen’s bound.

We begin by outlining the intuition behind Theorem 10. In a Jeffreys quantizer $J_K$, $D(b_k\|r_k)$ decreases with $k$ (see Lemma 5). Because of symmetry, $D(b_{k-1}\|r_k)$ increases with $k$. Therefore, the $\mathrm{WCR}$ occurs in the extreme bins. This is illustrated in Figure 3.3. In the majority of the bins, those away from the boundaries of the parameter space $\Omega$, $D(b_{k-1}\|r_k)$ and $D(b_k\|r_k)$ are similar and nearly constant w.r.t. $k$. Therefore, there is little room for improvement away from the boundaries of $\Omega$. On the other hand, near the boundaries of $\Omega$ there are some large WCBDs, and $J_K$ can be improved there.
Asymptotically, as $N$ increases, the regions where $\mathcal{J}_K$ can be improved become narrower, and the number of bins involved in these problematic regions is asymptotically negligible. This is also illustrated in Figure 3.3. Therefore, $D_K$, the WCD of the optimal quantizer with $K$ bins, will be similar to the WCBDs near the center of $\Omega$ obtained with $\mathcal{J}_K$. We will therefore construct an enhanced version of the Jeffreys quantizer with modified bin structures near the boundaries of $\Omega$. For optimum $K(N)$, this will produce a two-part code that achieves the lower bound of Theorem 10.

Before proving Theorem 10, we present a lemma that relates the smallest divergence (SD) and WCD of two different quantizers over an interval of parameters. Define $Q_{i}^{i+j} \triangleq \{b_k\}_{k=i}^{i+j}, \{r_k\}_{k=i+1}^{i+j}$ where $j > 0$, $b_k$ are not necessarily Jeffreys bin edges, and $r_k$ are not necessarily Jeffreys representation levels. Note that $Q_{i}^{i+j}$ is part of a quantizer that allocates $j$ bins (not $j+1$) to the interval $(b_i, b_{i+j})$. Define the SD of $Q_{i}^{i+j}$ in this interval by

\[ SD(Q_{i}^{i+j}) \triangleq \min_{k \in \{i+1, \ldots, i+j\}} \min \{D(b_{k-1}\|r_k), D(b_k\|r_k)\}. \]

Define the WCD of $Q_{i}^{i+j}$ in this interval by

\[ WD(Q_{i}^{i+j}) \triangleq \max_{k \in \{i+1, \ldots, i+j\}} \max \{D(b_{k-1}\|r_k), D(b_k\|r_k)\}. \]

There are other quantizers that also allocate $j$ bins to the interval $(b_i, b_{i+j})$. Define $TD_j(a, b)$ as the (theoretically) smallest WCD of any quantizer with $j$ bins over the interval $(a, b)$, i.e.,

\[ TD_j(a, b) \triangleq \min_{\{Q_{i}^{i+j}: b_i = a, b_{i+j} = b\}} WD(Q_{i}^{i+j}). \tag{3.29} \]

Define $Q^*_j(a, b)$ as a quantizer that achieves $TD_j(a, b)$, i.e.,

\[ Q^*_j(a, b) \triangleq \arg \min_{\{Q_{i}^{i+j}: b_i = a, b_{i+j} = b\}} WD(Q_{i}^{i+j}). \]

Lemma 6, which is proved in Appendix F, relates $SD(Q_{i}^{i+j})$, $WD(Q_{i}^{i+j})$, and $TD_j(b_i, b_{i+j})$. 72
Lemma 6: Any quantizer $Q^i_{i+j}$ that allocates $j$ bins to the interval $(b_i, b_{i+j})$ satisfies

$$SD(Q^i_{i+j}) \leq TD_j(b_i, b_{i+j}) \leq WD(Q^i_{i+j}).$$

With Lemma 6, we can prove Theorem 10.

Proof of Theorem 10: We begin by proving that any QP code has a WCR of at least $1.047 + O(N^{-0.25})$ bits above Rissanen’s lower bound. We then prove that the bound is tight by providing a construction that yields this performance.

Lower bound on WCR: Consider the Jeffreys quantizer $J_K$ with $K$ bins, and define

$$s = \left\lceil \frac{2}{\pi} \sqrt{K} \right\rceil.$$  

(3.30)

Define $\hat{\Omega} \triangleq (b_s, b_{K-s})$, and denote the $K - 2s$ bins of $J_K$ in the interval $\hat{\Omega}$ by $J_{K-2s}$. Recalling that in the Jeffreys quantizer $D(b_k || r_k)$ is monotone decreasing in $k$ (see Lemma 5), while $D(b_{k-1} || r_k)$ is monotone increasing in $k$, it follows that

$$SD(J_{K-2s}) = D(b_{s} || r_{s+1}).$$  

(3.31)

Consider a quantizer $Q_{K-2s} = Q_{0}^{K-2s}$ (with $K - 2s$ bins over the entire range $\Omega$). Because $\hat{\Omega} \subset \Omega$, there exists a quantizer $\hat{Q}_{1}^{K-2s+1}$ (with $K - 2s$ bins) such that $\hat{\Omega} = (\hat{b}_1, \hat{b}_{K-2s+1})$ and $WD(\hat{Q}_{1}^{K-2s+1}) \leq WD(Q_{0}^{K-2s}) = D_{Q_{K-2s}}$. Using (3.13),

$$R_{Q_{K-2s}}(N) = \log(K - 2s) + N \cdot D_{Q_{K-2s}}$$

$$\geq \log(K - 2s) + N \cdot WD(\hat{Q}_{1}^{K-2s+1}).$$  

(3.32)

Using Lemma 6,

$$SD(J_{K-2s}) \leq TD_{K-2s}(\hat{\Omega}) \leq WD(\hat{Q}_{1}^{K-2s+1}).$$  

(3.33)

Combining (3.31)–(3.33), we have

$$R_{Q_{K-2s}}(N) \geq \log(K - 2s) + N \cdot D(b_s || r_{s+1}).$$  

(3.34)

73
In Appendix G, we show that

$$D(b_j | r_{s+1}) \geq \frac{\pi^2}{8K^2 \ln(2)} + O\left(\frac{1}{K^{2.5}}\right). \quad (3.35)$$

Combining (3.34) and (3.35),

$$R_{Q_{K-2s}}(N) \geq \log(K - 2s) + \frac{N\pi^2}{8K^2 \ln(2)} + O\left(\frac{N}{K^{2.5}}\right)$$

$$= \log(K) + \frac{N\pi^2}{8K^2 \ln(2)} + O\left(\frac{N}{K^{2.5}}\right) + O\left(\frac{1}{\sqrt{K}}\right). \quad (3.36)$$

The right-hand side is minimized by choosing $K = K^*_C(N)$,

$$K^*_C(N) = \frac{\pi}{2} \sqrt{N} + O(N^{0.25}), \quad K^*_C(N) \in \mathbb{N}. \quad (3.37)$$

Substituting $K^*_C(N)$ into (3.36),

$$R_{Q_{K-2s}}(N) \geq \frac{1}{2} \log\left(\frac{\pi^2 e}{4} N\right) + O\left(\frac{1}{N^{0.25}}\right).$$

Comparing this to Rissanen’s bound (3.7), the additional redundancy that a code based on $Q_{K-2s}$ introduces is asymptotically lower bounded by

$$R_{Q_{K-2s}}(N) - R^2(N) \geq \frac{1}{2} \log\left(\frac{\pi^2 e}{4}\right) - \frac{1}{2} \log\left(\frac{\pi}{2}\right) + O(N^{-0.25})$$

$$= 1.047 + O(N^{-0.25})$$

bits. This concludes the proof of the lower bound.

**Construction of near-optimum code:** We will construct a two-part code, based on a quantizer $\tilde{J}_K$, whose WCR is $1.047 + O(N^{-0.25})$ bits above Rissanen’s bound. The main intuition is that, away from the boundaries of $\Omega$, the WCBDs of the Jeffreys quantizer $J_K$ are “equalized enough.” However, near the boundaries of $\Omega$ some of the WCBDs are large. Therefore, we choose $\tilde{J}_K$ to be identical to $J_K$ away from the boundaries of $\Omega$, but use enhanced bin structures near the boundaries of $\Omega$.

We now describe the precise structure of $\tilde{J}_K$, which is sketched in Figure 3.4. The quantizer $\tilde{J}_K$ consists of $K = K + 2s - 2$ bins, with bin edges $\tilde{b}_0, \ldots, \tilde{b}_K$ and representation
Figure 3.4 The constructed quantizer \( \tilde{J}_K \) and the Jeffreys quantizer \( J_K \).

levels \( \tilde{r}_1, \ldots, \tilde{r}_K \). The middle bins, i.e., bins \( 2s - 1, \ldots, K \), are identical to the bins \( s, \ldots, K - s + 1 \) of \( J_K \): for \( k \in \{s, \ldots, K - s + 1\} \), we have \( \tilde{b}_{k+s-2} = b_{k-1}, \tilde{b}_{k+s-1} = b_k \), and \( \tilde{r}_{k+s-1} = r_k \). The initial bins \( 1, \ldots, 2s-2 \) of \( \tilde{J}_K \) are an expanded version of bins \( 1, \ldots, s-1 \) of \( J_K \), such that each bin in \( J_K \) is split into two bins in \( \tilde{J}_K \). For \( k \in \{1, \ldots, s-1\} \) we have \( \tilde{b}_{2k-2} = b_{k-1}, \tilde{b}_{2k-1} = r_k + \epsilon, \tilde{b}_{2k} = b_k, \tilde{r}_{2k-1} = r_k \), and \( \tilde{r}_{2k} = b_k - \epsilon \), where \( \epsilon > 0 \) is small enough such that \( D(\tilde{b}_{2k-1}||\tilde{r}_{2k-1}) \) and \( D(\tilde{b}_{2k}||\tilde{r}_{2k}) \) are small (we call a WCBD small if it is smaller than all the WCBDs in the middle bins). Similarly, the end bins \( K+1, \ldots, K \) of \( \tilde{J}_K \) are an expanded version of bins \( K - s + 2, \ldots, K \) of \( J_K \). Because the edges and representation levels of the end bins are symmetric to those of the initial bins (Figure 3.4), we do not elaborate on the end bins.

We now compute the WCD of \( \tilde{J}_K \). We begin by showing that all the WCBDs of the initial and end bins are small, and then concentrate on the middle bins. For \( k \in \{1, \ldots, s-1\} \), we have

\[
D(\tilde{b}_{2k-2}||\tilde{r}_{2k-1}) = D(b_{k-1}||r_k),
\]

75
which is small because $D(b_{k-1}\|r_k)$ is monotone increasing. Next, $D(\tilde{b}_{2k-1}\|\tilde{r}_{2k-1})$ and $D(\tilde{b}_{2k}\|\tilde{r}_{2k})$ are small according to the selection criterion for $\epsilon$. Finally,

$$D(\tilde{b}_{2k-1}\|\tilde{r}_{2k}) = D(r_k + \epsilon \| b_k - \epsilon) < D(r_k\|b_k).$$

Using (3.22) and (3.23) (recall that $b_k$ and $r_k$ are defined for real-valued $k$), $r_k = b_{k-\frac{1}{2}}$ and $b_k = r_{k+\frac{1}{2}}$. But $D(b_k\|r_k)$ is nonincreasing in $k$ (Lemma 5), so $D(b_{k-1}\|r_k)$ is nondecreasing in $k$. Therefore, $D(b_{k-\frac{1}{2}}\|r_{k+\frac{1}{2}})$ is small, and so is $D(\tilde{b}_{2k-1}\|\tilde{r}_{2k})$. Because of symmetry, the same arguments can be applied to the expanded bins at the end of $\Omega$. We conclude that all the WCBDs in each expanded pair of bins in $\tilde{\mathcal{J}}_K$ are small. Therefore, we now concentrate on the WCBDs in the middle bins.

Recall that the structure of the middle of $\tilde{\mathcal{J}}_K$ is identical to that of the middle of $\mathcal{J}_K$. We thus consider $D(b_{k-1}\|r_k)$ and $D(b_k\|r_k)$ for $k \in \{s, \ldots, K - s + 1\}$. Because of symmetry, it suffices to consider $D(b_k\|r_k)$, and because this WCBD is monotone decreasing in $k$, we conclude that all WCBDs of the middle of the quantization structure are upper bounded by $D(b_s\|r_s)$. Because all WCBDs of the expanded pairs of bins are small, all WCBDs of $\tilde{\mathcal{J}}_K$ are upper bounded by $D(b_s\|r_s)$, so

$$D_{\tilde{\mathcal{J}}_K} = D(b_s\|r_s).$$

In Appendix G, we show that

$$D(b_s\|r_s) \leq \frac{\pi^2}{8K^2 \ln(2)} + O\left(\frac{1}{K^{2.5}}\right),$$

where the constants in the $O(\cdot)$ term are different from those in (3.35).
We now compute the $\overline{\text{WCR}}$ of the code that uses $\tilde{J}_K$. Using (3.13), (3.38) and the definition $\tilde{K} = K + 2s - 2$,

$$R_{\tilde{J}_K}(N) = \log(\tilde{K}) + N \cdot D(b_s || r_s) \leq \log(K + 2s - 2) + \frac{N\pi^2}{8K^2 \ln(2)} + O\left(\frac{N}{K^{2.5}}\right) = \log(K) + \frac{N\pi^2}{8K^2 \ln(2)} + O\left(\frac{N}{K^{2.5}}\right) + O\left(\frac{1}{\sqrt{K}}\right).$$

Because this result is identical to (3.36) except for the constants in the $O(\cdot)$ terms, the construction of $\tilde{J}_K$ with $K^*_c(N)$ from (3.37) provides a $\overline{\text{WCR}}$ that is $1.047 + O(N^{-0.25})$ bits above Rissanen’s bound. \hfill \Box

**Remark 9** With (3.26) and (3.37) we have

$$K^*_c(N) \approx 1.571\sqrt{N} < K^*_c(N) \approx 1.772\sqrt{N}.$$ 

This can be explained by noticing that, for fixed $K$, $D_{\tilde{J}_K}$ is larger than the WCD of the constructed quantizer $\tilde{J}_K$. Because the WCD decreases with $K$, the tradeoff between the coding lengths of the first and second parts of the code in (3.13) is biased towards using more bins with $\tilde{J}_K$.

With Theorem 10, which provides an attainable lower bound on the $\overline{\text{WCR}}$ of QP codes, we can prove a similar result for the lowest attainable WCR of QP codes. The proof appears in Appendix H.

**Theorem 11** The lowest WCR of any QP code is $1.047 + O(N^{-0.25})$ bits above Rissanen’s bound.

### 3.5.3 Modified Jeffreys method

Although the proof of Theorem 10 provides a code with asymptotically optimum $\overline{\text{WCR}}$ and WCR (1.047 bits above Rissanen’s bound), the construction is awkward. We
now provide a modified version of the Jeffreys code that asymptotically has the same WCR and WCR, by decreasing the discrepancy between $D(b_{k-1}||r_k)$ and $D(b_k||r_k)$ near the boundaries of $\Omega$. Towards this end, we use the bin edges of the Jeffreys quantizer, and use (3.27) to calculate the representation levels. Therefore,

$$r_k = \frac{1}{\mu(\sin^2(\frac{\pi}{2K})) - \mu(\sin^2(\frac{\pi(2k-1)}{2K}))}. \frac{\sin^2(\frac{\pi k}{2K}) - \sin^2(\frac{\pi k}{2K})}{1 + 2 \sin^2(\frac{\pi k}{2K}) - \sin^2(\frac{\pi(2k-1)}{2K})}$$

We call this a modified Jeffreys quantizer.

With the modified Jeffreys method, the (equalized) WCBDs of bins near the boundaries of $\Omega$ become lower than the WCBDs away from the boundaries. Therefore, the WCD of the modified Jeffreys quantizer is determined by the WCBDs of the middle bins. The derivation of the WCD and optimization of $K$ are similar to those in the proof of Theorem 10. With the optimal $K$, which is approximately $\frac{\pi}{2} \sqrt{N}$, we obtain the modified Jeffreys code.

**Conjecture 2** The WCR and WCR of the modified Jeffreys code are $1.047 + o(1)$ bits above Rissanen’s bound.

For practical purposes, the modified Jeffreys code is as simple and computationally efficient as the Jeffreys code. However, its WCR and WCR are asymptotically as good as any other QP code. Figure 3.5 compares the WCRs of Jeffreys codes, modified Jeffreys codes, quantizer-optimal codes, and the WCR of operational Jeffreys codes.

### 3.6 Finite Alphabets

Till now, we considered Bernoulli sequences over a binary alphabet. In this section, we generalize to i.i.d. sequences over a finite alphabet $\mathcal{X} = \{0, 1, \ldots, |\mathcal{X}| - 1\}$. We begin by computing Rissanen’s bound for a finite alphabet in Section 3.6.1. Then, Section 3.6.2
Figure 3.5 The WCR above Rissanen’s bound of the Jeffreys code approaches 1.221 bits. The WCR of quantizer-optimal codes and modified Jeffreys codes, and the WCR of operational Jeffreys codes, approach 1.047 bits above Rissanen’s bound.

proposes several two-part codes for finite alphabets. Finally, Section 3.6.3 improves the WCR by adding priors on the quantization bins. A different design approach to quantizer construction was provided by Chou et al. [52].

3.6.1 Rissanen’s bound for finite alphabets

We now have a vector parameter $\Theta$, consisting of $|\mathcal{X}| - 1$ symbol frequencies,

$$\Theta = (\theta^0, \theta^1, \ldots, \theta^{|\mathcal{X}|-2}) ,$$

and the frequency of the last symbol, $\theta^{|\mathcal{X}|-1}$, satisfies

$$\theta^{|\mathcal{X}|-1} = 1 - \sum_{j=0}^{|\mathcal{X}|-2} \theta^j.$$
Recall from (3.6) that we denote Rissanen’s bound on the WCR for length-$N$ i.i.d. sequences over a size-$|\mathcal{X}|$ alphabet by $R^{[\mathcal{X}]}(N)$. The Fisher information is \cite{28}
\[
I(\Theta) = \prod_{j=0}^{|\mathcal{X}|-1} \frac{1}{\theta^j},
\]
and the parameter set $\Omega$ is the $|\mathcal{X}| - 1$ dimensional probability simplex. Theorem 12, which computes Rissanen’s bound for arbitrary alphabet sizes explicitly, was provided by Rissanen \cite{28} without proof. For completeness, we prove it in Appendix I.

**Theorem 12** Rissanen’s lower bound on the WCR is
\[
R^{[\mathcal{X}]}(N) = \frac{|\mathcal{X}|-1}{2} \log \left( \frac{N}{2} \right) + \log \left( \frac{\sqrt{\pi}}{\Gamma \left( \frac{|\mathcal{X}|}{2} \right)} \right) + o(1) \tag{3.40}
\]
bits, where $\Gamma(\cdot)$ is the gamma function.

Using Stirling’s formula $\Gamma(t) \approx \sqrt{2\pi/t}(t/e)^t$ yields the following approximation for large $|\mathcal{X}|$:
\[
R^{[\mathcal{X}]}(N) \approx \frac{|\mathcal{X}|-1}{2} \log \left( \frac{N e}{|\mathcal{X}|} \right) + \frac{1}{2} \log \left( \frac{e}{2} \right) \tag{3.41}
\]
bits. (For $|\mathcal{X}| = 3$, the error in the approximation is 0.079 bits; for $|\mathcal{X}| = 50$, the error is 0.005 bits.) An important conclusion from Theorem 12 is that for fixed $N$, as $|\mathcal{X}|$ is increased, each additional parameter adds less to the WCR.

**3.6.2 Two-part codes for finite alphabets**

Just as the NML code has the smallest WCR for the binary alphabet, a similar result can be obtained for nonbinary alphabets. Instead, we use two-part codes for encoding i.i.d. sequences over finite alphabets, and evaluate their WCRRs and WCRs.

**Separable scalar quantization:** With an alphabet $\mathcal{X}$, we need to describe $|\mathcal{X}| - 1$ ML parameter estimates. A trivial implementation would quantize the parameters $\theta^0, \theta^1, \ldots, \theta^{|\mathcal{X}|-2}$ independently of each other using separable scalar quantization, and
then encode \( x \) accordingly. However, separable quantization does not always work well. For example, consider \(|\mathcal{X}| = 3\), and suppose that \( \theta^0 + \theta^1 = 1 - \epsilon \) for \( 0 \leq \epsilon \ll 1 \), and that \( \theta^0 \) and \( \theta^1 \) are away from the boundaries of the probability simplex. We have seen that, for \(|\mathcal{X}| = 2\), good quantizers are relatively coarse away from the boundaries of \( \Omega \). Therefore, because \( \theta^0 \) and \( \theta^1 \) are away from the boundaries of \( \Omega \), the (separable) quantization of \( \theta^0 \) and \( \theta^1 \) will be relatively coarse, so the implied quantization of \( \theta^2 \) will also be coarse. But \( \theta^2 = \epsilon \), so it requires fine quantization because divergences near the boundaries of the probability simplex fluctuate rapidly. Therefore, separable scalar quantization may yield high redundancies near the boundaries of the probability simplex.

**Encoding with normalized parameters:** Define \( n^j(x) \triangleq \sum_{i=1}^{N} 1 \{ x_i = j \} \), the number of symbols in \( x \) equal to \( j \), and \( N^j(x) \triangleq \sum_{k=j}^{\mid \mathcal{X} \mid - 1} n^k(x) \), the number of symbols in \( x \) equal to \( j \) or greater. To simplify the presentation in the remainder of Section 3.6, we use the notation \( \Theta \) instead of \( \Theta_{ML}(x) \), i.e., \( n^j(x) = N \cdot \theta^j \) and \( N^j(x) = N \sum_{k=j}^{\mid \mathcal{X} \mid - 1} \theta^k \). Define

\[
v^j \triangleq p(\text{symbol is } j | \text{symbol is not } 0, 1, \ldots, j - 1) = \frac{\theta^j}{1 - \sum_{k=0}^{j-1} \theta^k} = \frac{n^j(x)}{N^j(x)} \tag{3.42}
\]

for \( j \in \{1, \ldots, \mid \mathcal{X} \mid - 1\} \), and \( v^0 \triangleq p(\text{symbol is } 0) = \theta^0 \). We now have

\[
\theta^j = \left[ \prod_{k=0}^{j-1} (1 - v^k) \right] v^j. \tag{3.43}
\]

We call \( \{v^j\}_{j=0}^{\mid \mathcal{X} \mid - 2} \) *normalized parameters* because each \( v^j \) may be as large as 1, no matter what \( \{u^k\}_{k=0}^{j-1} \) were. Furthermore, if \( \Theta \) is near the boundary of the probability simplex, then \( \exists j \in \{0, 1, \ldots, \mid \mathcal{X} \mid - 2\} \) such that \( v^j \) is near 0 or 1. By quantizing \( \{v^j\}_{j=0}^{\mid \mathcal{X} \mid - 2} \) instead of \( \{\theta^j\}_{j=0}^{\mid \mathcal{X} \mid - 2} \), normalized parameters will be finely quantized if necessary.

We now describe a generic two-part code that uses normalized parameters. In the first part of the code, each \( v^j \) is quantized to \( \tilde{v}^j \), \( j \in \{0, \ldots, \mid \mathcal{X} \mid - 2\} \). We describe several
different quantization schemes for the first part of the code later. Based on (3.43), in the second part of the code the probability assigned to each symbol \( x_i = j \in \{0, \ldots, |\mathcal{X}| - 1\} \) is

\[
p(x_i = j) = \tilde{\theta}^j \triangleq \left[ \prod_{k=0}^{j-1} (1 - \tilde{v}^k) \right] \tilde{v}^j,
\]

(3.44)

where \( \tilde{v}^{|\mathcal{X}|-1} = 1 \) because \( v^{|\mathcal{X}|-1} = 1 \) (3.42). It is readily verified that

\[
\tilde{\theta}^{|\mathcal{X}|-1} = 1 - \sum_{j=0}^{|\mathcal{X}|-2} \tilde{\theta}^j,
\]

so (3.44) is a PMF. We call the second part of the two-part code based on the probability assignment of (3.44) encoding with normalized parameters. We denote the probability assigned to \( x \) when encoding with normalized parameters by \( p_n(x) \), i.e.,

\[
p_n(x) = \prod_{i=1}^{N} \tilde{\theta}^{x_i}.
\]

We also denote the coding length corresponding to \( p_n(x) \) by \( l_n(x) = -\log(p_n(x)) \), and the redundancy of the second part of the code when encoding \( x \) with normalized parameters by \( R_n(x) = l_n(x) - l_{ML}(x) \). Lemma 7, which evaluates \( R_n(X) \), is proved in Appendix J.

**Lemma 7** The redundancy of encoding with normalized parameters satisfies

\[
R_n(x) = \sum_{j=0}^{|\mathcal{X}|-2} N^j(x) D(v^j \| \tilde{v}^j).
\]

(3.45)

**Separable scalar quantization with normalized parameters:** Earlier, we described encoding with normalized parameters as the second part of a generic two-part code. One simple possibility for the first part of the two-part code consists of separately quantizing each \( v^j = n^j(x) / N^j(x) \) to \( \tilde{v}^j \), \( j \in \{0, \ldots, |\mathcal{X}| - 2\} \) using an identical scalar quantizer with \( K \) bins (the same quantizer for all \( |\mathcal{X}| - 1 \) parameters). We consider more sophisticated methods for the first part of the code in Section 3.6.3.

In particular, consider using quantizer \( Q_K \), as previously defined for encoding Bernoulli sequences, for quantizing each of the \( |\mathcal{X}| - 1 \) parameters in the first part of the code. The
second part of the code consists of encoding $x$ with the (quantized) normalized parameters. We call this two-part code *separable scalar quantization with normalized parameters and an underlying Bernoulli quantizer* $Q_K$. Denote the WCR of separable scalar quantization with an underlying Bernoulli quantizer $Q_K$ for a size-$|\mathcal{X}|$ alphabet by $R^{[\mathcal{X}]}_{Q_K}(N)$. This WCR comprises both the explicit description of the parameters and the redundancy of encoding with normalized parameters, which was evaluated in Lemma 7. For $|\mathcal{X}| = 2$, one parameter $\theta^0$ is quantized, and encoding with normalized parameters is identical to the second part of a Bernoulli two-part code, because $v^0 = \theta^0$. Therefore, we have the encouraging result $R^2_{Q_K}(N) = R_{Q_K}(N)$. We now provide a result for the more general case.

**Theorem 13** Using separable scalar quantization with normalized parameters and an underlying Bernoulli quantizer $Q_K$, the WCR for length-$N$ i.i.d. sequences over a size-$|\mathcal{X}|$ alphabet satisfies

$$R^{[\mathcal{X}]}_{Q_K}(N) \leq (|\mathcal{X}| - 1)R_{Q_K}(N).$$

**Proof:** The WCR consists of $(|\mathcal{X}| - 1)\log(K)$ bits required for the explicit description of the parameters in the first part of the two-part code, and the redundancy of encoding with normalized parameters, which was evaluated in Lemma 7. Therefore, because $N^j(x) \leq N$,

$$R^{[\mathcal{X}]}_{Q_K}(N) \leq (|\mathcal{X}| - 1)\log(K) + N \sum_{j=0}^{[\mathcal{X}]-2} D(v^j \| \tilde{v}^j) \leq (|\mathcal{X}| - 1) [\log(K) + N \cdot D_{Q_K}] = (|\mathcal{X}| - 1)R_{Q_K}(N),$$

(3.46)

(3.47)

where (3.46) follows from Lemma 7 and (3.47) follows (3.13). \hfill $\square$

Using Theorem 10, $R^{[\mathcal{X}]}_{Q_K}(N) \leq (|\mathcal{X}| - 1) \left[ \frac{1}{2} \log \left( \frac{2}{N} \epsilon N \right) + O \left( \frac{1}{N^{1/4}} \right) \right]$ bits. However, using (3.41), the additional redundancy above Rissanen’s bound is approximately
\[
\frac{|X|}{2} \log \left( \frac{e^{|X|}}{4} \right) - \frac{1}{2} \log \left( \frac{5}{2} \right) \text{ bits. We will now alleviate this superlinear additional redundancy.}
\]

### 3.6.3 Two-part codes with priors

Separable scalar quantization with normalized parameters uses the same quantizer for each of the parameters. That quantizer was optimized for length-\(N\) Bernoulli sequences, but Lemma 7 indicates that \(\hat{\nu}^j\) only contributes an \(N^j(x)D(v^j||\hat{\nu}^j)\) term to the redundancy, not \(N \cdot D(v^j||\hat{\nu}^j)\). Using (3.13), this divergence term appears in the redundancy formula for length-\(N^j(x)\) Bernoulli sequences. Therefore, it makes sense to quantize \(\hat{\nu}^j\) with a quantizer optimized for length-\(N^j(x)\) sequences. Because

\[
N^j(x) \approx N \prod_{k=0}^{j-1} (1 - \hat{\nu}^k),
\]

in the first part of the code we can sequentially quantize \(v^0, \ldots, v^{|X| - 2}\), where each \(v^j\) is quantized with a quantizer optimized for length-\(N^j(x)\) sequences, and \(N^j(x)\) is estimated by (3.48). However, the reduction in coding length is a function of the parameter values; if all the symbols of \(x\) are \(|X| - 1\), i.e., \(n_e^{\log|x|-1}(x) = N\), then \(N^j(x) = N\) for all \(j \in \{0, \ldots, |X| - 1\}\), there is no savings in coding length; hence, the additional \(\overline{\text{WCR}}\) above Rissanen’s bound is still superlinear in \(|X| - 1\). In the remainder of the subsection we introduce yet another improvement to the latter scheme, which yields a scheme whose redundancy above Rissanen’s bound is linear in \(|X| - 1\).

We reiterate that the lowest achievable WCR is attained by the NML code, which encodes all sequences with the same redundancy (3.4). Although we do not replicate the WCR of the NML code, we can improve our WCR by striving to attain similar redundancies for all input sequences. We first describe the structure of two-part codes with priors, and then construct a code with priors whose WCR is \((|X| - 1)[1.047 + o(1)]\) bits above Rissanen’s bound.
Two-part codes with priors use encoding with normalized parameters in their second part. The first part is different; instead of describing the bin that a parameter belongs to using \( \log(K) \) bits, two-part codes with priors use a (nonuniform) prior to describe the bin; hence the length of the description is not constant. When quantizing \( v^j \) sequentially, we have already described the quantized parameters \( \tilde{v}^0, \ldots, \tilde{v}^{j-1} \), so \( N^j(x) \) can be estimated with (3.48). We then generate a quantizer \( Q_{K^j} \) with \( K^j \) bins, and determine in which of \( K^j \) bins \( v^j \) lies. Denoting the index of that bin by \( k^j \), we have \( v^j \in [b_{k^j-1}, b_{k^j}) \), where \( b_{k^j-1} \) and \( b_{k^j} \) are bin edges of \( Q_{K^j} \) (not necessarily Jeffreys bin edges). Two-part codes with priors describe the bin with a prior

\[
p(v^j \in [b_{k^j-1}, b_{k^j})) \triangleq \frac{p_{k^j}}{\psi^j}, \tag{3.49}
\]

where \( p_{k^j} > 0 \) is a function of the bin index, and \( \psi^j \) is a normalization constant, i.e.,

\[
\psi^j \triangleq \sum_{k^j=1}^{K^j} p_{k^j}. \tag{3.50}
\]

By letting the prior compensate for the decrease in \( N^k(x) \), \( k \in \{j+1, \ldots, |\mathcal{X}|-1\} \) caused by large \( \tilde{v}^j \), which also decreases the coding length required for symbols that are greater than \( j \), we can achieve similar redundancies no matter what the parameters are.

We now construct a code whose additional WCR above Rissanen’s bound is linear in \( |\mathcal{X}| - 1 \). The analysis of the WCR is provided in Theorem 14. In the first part of the code, we use quantizers based on the construction \( \tilde{J}_K \) from the proof of Theorem 10. We use \( \tilde{K}^j = K^j + 2s^j - 2 \), bin edges \( \{\tilde{b}_k\}_{k=0}^{\tilde{K}^j} \), and representation levels \( \{\tilde{r}_k\}_{k=1}^{\tilde{K}^j} \), where \( K^j = \left\lceil \frac{2}{\sqrt{N^j(x)}} \right\rceil \) \( (N^j(x) \) is estimated with (3.48)) and \( s^j = \left\lceil \frac{2}{\sqrt{K^j}} \right\rceil \). We define the prior (3.49) as

\[
p_{k^j} \triangleq (1 - \tilde{b}_{k^j-1})^{\frac{|\mathcal{X}| - j - 2}{2}}, \tag{3.51}
\]

where bin \( k^j \) is \( [\tilde{b}_{k^j-1}, \tilde{b}_{k^j}) \). We denote the coding length assigned to \( x \) by this construction by \( l^{|\mathcal{X}|}(x) \), define the redundancy of the code by \( R_{c}^{|\mathcal{|X|}}(x) \triangleq l^{|\mathcal{X}|}(x) - l_{ML}(x) \), and
define the WCR for length-$N$ i.i.d. sequences over a size-$|\mathcal{X}|$ alphabet by $R_c^{[X]}(N) \triangleq \max_{x \in \mathcal{X}^N} \{R_c^{[X]}(x)\}$.

**Theorem 14** The WCR of the two-part code with priors constructed above satisfies

$$R_c^{[X]}(N) \leq R^{[X]}(N) + (|\mathcal{X}| - 1)(1.047 + o(1)).$$

**Proof:** The proof is by induction on $|\mathcal{X}|$. The basis of the induction is $|\mathcal{X}| = 2$. We need to prove that, for each $N \in \mathbb{N}$, $R_c^2(N) = R^2(N) + 1.047 + o(1)$. In the first part of the code we quantize and encode $v^0 = \theta^0$. We have $N^0(x) = N$, the prior and normalization constant satisfy $p_{k0} = 1$ (3.51) and $\psi^0 = \tilde{K}^0$ (3.50), so we encode $k^0$ with $\log(\tilde{K}^0)$ bins, just like the construction in the proof of Theorem 10. In the second part of the code, encoding with normalized parameters using the representation levels of $\tilde{J}_K$ is identical to the second part of the construction in the proof of Theorem 10. Therefore, we have $R_c^2(N) \leq R^2(N) + 1.047 + o(1)$.

Assuming that the result holds for $|\mathcal{X}|$ for each $N \in \mathbb{N}$, we need to prove that $R_c^{[X]+1}(N) \leq R^{[X]+1}(N) + |\mathcal{X}|(1.047 + o(1))$ for each $N \in \mathbb{N}$. The coding length $l_c^{[X]+1}(x)$ comprises $|\mathcal{X}|$ terms in the first part of the code and $l_n(x)$, the length of encoding with normalized parameters, in the second part. Combining (3.45) and (3.49),

$$R_c^{[X]+1}(x) = \sum_{j=0}^{|X|-1} \log \left( \frac{\psi^j}{p_{kJ}} \right) + \sum_{j=0}^{|X|-1} N_j(x)D(v^j||\bar{v}^j); \quad (3.52)$$

hence,

$$l_c^{[X]+1}(x) = \sum_{j=0}^{|X|-1} \log \left( \frac{\psi^j}{p_{kJ}} \right) + \sum_{j=0}^{|X|-1} N_j(x)D(v^j||\bar{v}^j) + \sum_{j=0}^{|X|} n^j(x) \log \left( \frac{N}{n^j(x)} \right). \quad (3.53)$$

We now partition our construction, which we call Code A, into two codes, which we call Code B and Code C. In Code B, we quantize $v^0$ to $\bar{v}^0$, encode $\bar{v}^0$ using $p_{k0}$, and describe which symbols of $x$ are 0 by encoding with the normalized parameter $\bar{v}^0$. The
coding length $l_B(x)$ of Code B satisfies

$$l_B(x) = \log \left( \frac{\psi_0}{p_{k^0}} \right) + N \cdot \mathcal{D}(v^0 \| \bar{v}^0) + n^0(x) \log \left( \frac{N}{n^0(x)} \right) + N^1(x) \log \left( \frac{N}{N^1(x)} \right). \quad (3.54)$$

We denote by $\tilde{x}$ the length-$N^1(x)$ sequence comprised of the symbols of $x$ that are greater than 0, and by $\hat{x}$ the sequence obtained by changing each appearance of the symbol $j$ in $\tilde{x}$ to $\tilde{j} \triangleq j - 1$. Note that $\hat{x}$ is a length-$N^1(x)$ sequence over a size-$|\mathcal{X}|$ alphabet, $n^{\tilde{j}}(\hat{x}) = n^j(x)$, and $N^{\hat{j}}(\hat{x}) = N^j(x)$ for $\tilde{j} \in \{0, \ldots, |\mathcal{X}| - 1\}$. After using Code B, it suffices to describe $\hat{x}$ in order to be able to reconstruct $x$.

In Code C, $\tilde{x}$ is described using the construction for length-$N^1(x)$ sequences over size-$|\mathcal{X}|$ alphabets. Using (3.53), the coding length $l_C(x)$ of Code C satisfies

$$l_C(x) = \sum_{j=0}^{|\mathcal{X}|-2} \log \left( \frac{\psi^j}{p_{k^j}} \right) + \sum_{j=0}^{|\mathcal{X}|-2} N^{\tilde{j}}(\tilde{x}) \mathcal{D}(v^{\tilde{j}} \| \bar{v}^{\tilde{j}}) + \sum_{j=0}^{|\mathcal{X}|-1} n^{\hat{j}}(\hat{x}) \log \left( \frac{N^1(x)}{n^j(\hat{x})} \right).$$

Owing to the structure of $\tilde{x}$, we have $n^{\tilde{j}}(\hat{x}) = n^j(x)$, $N^{\hat{j}}(\hat{x}) = N^j(x)$, $\hat{K}^j$ of Code C equals $\hat{K}^j$ of Code A, $p_{k^j}$ of Code C equals $p_{k^j}$ of Code A, $\psi^j$ of Code C equals $\psi^j$ of Code A, $v^j$ of Code C equals $v^j$ of Code A, and $\bar{v}^j$ of Code C equals $\bar{v}^j$ of Code A. Therefore,

$$l_C(x) = \sum_{j=1}^{|\mathcal{X}|-1} \log \left( \frac{\psi^j}{p_{k^j}} \right) + \sum_{j=1}^{|\mathcal{X}|-1} N^j(x) \mathcal{D}(v^j \| \bar{v}^j) + \sum_{j=1}^{|\mathcal{X}|} n^j(x) \log \left( \frac{N^1(x)}{n^j(x)} \right). \quad (3.55)$$

Using the inductive hypothesis, the redundancy $R_C(x)$ of Code C satisfies

$$R_C(x) = \sum_{j=1}^{|\mathcal{X}|-1} \log \left( \frac{\psi^j}{p_{k^j}} \right) + \sum_{j=1}^{|\mathcal{X}|-1} N^j(x) \mathcal{D}(v^j \| \bar{v}^j) \leq R_{|\mathcal{X}|}(N^1(x)) + (|\mathcal{X}| - 1)[1.047 + o(1)]. \quad (3.56)$$

Combining (3.53), (3.54), and (3.55),

$$l_B(x) + l_C(x) = \sum_{j=0}^{|\mathcal{X}|-1} \log \left( \frac{\psi^j}{p_{k^j}} \right) + \sum_{j=0}^{|\mathcal{X}|-1} N^j(x) \mathcal{D}(v^j \| \bar{v}^j)$$

$$+ n^0(x) \log \left( \frac{N}{n^0(x)} \right) + N^1(x) \log \left( \frac{N}{N^1(x)} \right) + \sum_{j=1}^{|\mathcal{X}|} n^j(x) \log \left( \frac{N^1(x)}{n^j(x)} \right)$$

$$= l^{|\mathcal{X}|+1}(x). \quad (3.57)$$

87
Combining (3.52), (3.56), and (3.57),
\[
R^{|X|+1}_c(N) \leq \log \left( \frac{\psi^0}{p_{k^0}} \right) + N^0(x)D(v^0||\bar{s}^0) + R^{|X|}(N^1(x)) + (|X| - 1)[1.047 + o(1)]. \tag{3.58}
\]

We now compute \( \psi^0 \).
\[
\psi^0 = \sum_{k^0=1}^{K^0} p_{k^0} = \sum_{k^0=1}^{K^0} (1 - \tilde{b}_{k^0-1})^{\frac{|X|-1}{2}}
\]
\[
= O(s^0) + \sum_{k^0=2}^{K^0} (1 - \tilde{b}_{k^0-1})^{\frac{|X|-1}{2}} \tag{3.59}
\]
\[
= O(s^0) + \sum_{k=s^0}^{K^0-1} (1 - b_k)^{\frac{|X|-1}{2}} \tag{3.60}
\]
\[
= O(s^0) + \sum_{k=s^0}^{K^0-s^0+1} (b_k)^{\frac{|X|-1}{2}} \tag{3.61}
\]
\[
= O(s^0) + \sum_{k=s^0}^{K^0-s^0+1} \left[ \sin \left( \frac{\pi k}{2K^0} \right) \right]^{\frac{|X|-1}{2}} \tag{3.62}
\]
\[
= O(\sqrt{K^0}) + \frac{2K^0}{\pi} \left[ \int_{\frac{1}{\sqrt{K^0}}}^{\frac{\pi}{K^0}} \sin(\theta)^{\frac{|X|-1}{2}} d\theta + o(1) \right] \tag{3.63}
\]
\[
= \sqrt{N} \left[ \int_0^{\frac{\pi}{2}} \cos(\theta)^{|X|-1} d\theta + o(1) \right], \tag{3.64}
\]

where (3.59) enables us to concentrate on the middle bins (Figure 3.4), (3.60) uses the relationship between bins of \( \tilde{J}^0_k \) and \( J^0_k \) from the proof of Theorem 10, (3.61) uses the symmetry relation \( b_k = 1 - b_{K-k} \) (Jeffreys bin edges), (3.62) uses (3.22), (3.63) uses the definition of \( s^0 \) and Riemann integration, and (3.64) uses the definition of \( K^0 \) and the identity \( \int_0^{\pi/2} \sin(\theta)^k d\theta = \int_0^{\pi/2} \cos(\theta)^k d\theta \). Using (3.39) and the definitions of \( K^0 \) and \( s^0 \),
\[
N^0(x)D(v^0||\bar{s}^0) \leq \frac{1}{2\ln(2)} + o(1) = \frac{1}{2} \log(e) + o(1). \tag{3.65}
\]

Combining (3.51), (3.58), (3.64), and (3.65),
\[
R^{|X|+1}_c(N) \leq \frac{1}{2} \log(N) + \log \left[ \int_0^{\frac{\pi}{2}} \cos(\theta)^{|X|-1} d\theta \right] - \frac{|X| - 1}{2} \log(1 - \tilde{b}_{k^0-1}) + \frac{1}{2} \log(e)
\]
\[
+ R^{|X|}(N^1(x)) + (|X| - 1)[1.047 + o(1)].
\]

88
Because \( v^0 \geq \tilde{b}_{k_0-1} \), \( N^1(x) = N(1 - v^0) \leq N(1 - \tilde{b}_{k_0-1}) \), so using (3.6) we have

\[
R^{|\mathcal{X}|}(N^1(x)) - \frac{|\mathcal{X}| - 1}{2} \log(1 - \tilde{b}_{k_0-1}) \leq R^{|\mathcal{X}|}(N).
\]

Also, using (3.6) and (I.3),

\[
R^{|\mathcal{X}|+1}(N) = R^{|\mathcal{X}|}(N) + \frac{1}{2} \log \left( \frac{N}{2\pi} \right) + \log \left[ 2 \int_0^{\frac{\pi}{2}} \cos(\theta)|\mathcal{X}|^{\frac{1}{2}} d\theta \right] + o(1).
\]

Therefore,

\[
R_c^{|\mathcal{X}|+1}(N) \leq R^{|\mathcal{X}|+1}(N) + \frac{1}{2} \log \left( \frac{\pi e}{2} \right) + \left( |\mathcal{X}| - 1 \right) [1.047 + o(1)]
\]

\[
= R^{|\mathcal{X}|+1}(N) + |\mathcal{X}|[1.047 + o(1)],
\]

which completes the proof. \( \square \)

**Remark 10** The construction used here for i.i.d. sequences is based on the construction \( \hat{J}_K \) from the proof of Theorem 10. Asymptotically, the same WCR performance can be obtained by using the modified Jeffreys code.

The redundancy above Rissanen’s bound scales linearly with \(|\mathcal{X}| - 1\) because our approach essentially quantizes each parameter with the first part of a QP code that was derived for Bernoulli sequences; using a prior on the bins mitigates the superlinear redundancy over Rissanen’s bound.

### 3.7 Discussion

We studied quantized-parameter (QP) two-part codes, which are relatively simple to implement and have applications in distributed universal lossless source coding. In the first part of the code, after computing \( \theta_{ML}(x) \), the representation level \( \tilde{\theta}_{ML}(x) \) is computed once, and its index is encoded with \( \log(K) \) bits. In the second part of the code, \( \tilde{\theta}_{ML}(x) \) is used for assigning a probability to \( x \). Using a quantizer motivated by

89
Jeffreys’ prior, where uniform quantization in the $\eta(\cdot)$ domain provides similar worst-case bin divergences in the different bins, we upper bounded the worst-case redundancy (WCR) by $1.221 + O(\frac{1}{N})$ bits above Rissanen’s bound.

We then proved that the best achievable WCR among QP Bernoulli codes is $1.047 + O(N^{-0.25})$ bits above Rissanen’s bound. We also proposed a modified Jeffreys code that essentially achieves the lower bound. Finally, we discussed how two-part codes can be used for universal encoding of i.i.d. sequences over finite alphabets.

This chapter concentrated on minimizing the WCR using scalar quantizers. An interesting problem involves finding better two-part codes for i.i.d. sequences over finite alphabets. While our approach essentially quantized each parameter with the first part of a QP code that was derived for Bernoulli sequences, it may be possible to significantly improve these results with vector quantizers [52]. An alternative to the WCR approach is to assume some prior on the Bernoulli parameter $\theta$. That leads to a setup in which we minimize the expected redundancy. A potentially interesting problem would be to compare two-part codes that minimize the expected redundancy with the codes studied in this chapter.

Finally, this chapter provided a direction for distributed lossless source coding with low redundancies. There may be other coding methods that can be used for distributed compression, while still maintaining low worst-case redundancies. We leave these for future research.
CHAPTER 4

A WORK-EFFICIENT PARALLEL TWO-PASS MDL CONTEXT TREE ALGORITHM FOR UNIVERSAL SOURCE CODING

4.1 Motivation and Background

4.1.1 Motivation

At present, typical lossless data compression algorithms, which are implemented in software, run at least an order of magnitude slower than the throughput delivered by hard disks; their performance is even worse when compared to optical communication devices. In computer and communication systems, the maximal data rate that can be supported is limited by the slowest component. Therefore, in many systems lossless coding is the bottleneck, and cannot be used.

One obvious approach to speed up compression algorithms is to implement them in special-purpose hardware. Although this approach may accelerate compression by approximately an order of magnitude, there are still many systems where this does not suffice. Ultimately, in order for lossless compression to become appealing for a broader range of applications, we must concentrate less on implementation details and more on sophisticated new algorithms.
Parallelization is a possible direction for fast source coding algorithms. By compressing in parallel, either in dedicated hardware or in software on a multiprocessor computer, we may obtain algorithms that are faster by orders of magnitude. However, with naive parallelization, which consists of partitioning the original input into $B$ blocks and processing each block independently of the other blocks, increasing $B$ degrades the compression quality (numerical data that supports this claim appears in [55]). Therefore, such methods, which process blocks independently in parallel, have limited potential.

In this chapter, we present a novel two-pass minimum description length (MDL) source coding algorithm that coordinates multiple computational units running in parallel, such that the compression loss incurred by using increasingly more computational units is insignificant. Our algorithm compresses $B$ length-$(N/B)$ blocks in parallel with $O(N/B)$ time complexity. This work-efficient [29] algorithm is our most significant contribution. In the remainder of Section 4.1, we provide some background and lay out a design space for parallel compression. In Section 4.2, we derive some mathematical details of our parallel two-pass MDL algorithm. Section 4.3 works out the algorithmic details, and Section 4.4 presents some of the properties of the algorithm, including the compression loss caused by the parallelization. Then, in Section 4.5 we discuss several extensions of our parallel algorithm. Finally, Section 4.6 discusses our results and offers several topics for future research.

### 4.1.2 Design space

The design space for parallel algorithms is richer than that for serial algorithms; hence we lay out some parameters of interest.

- *Compression quality* - algorithms with poorer compression are in general simpler.
• *Latency* - the amount of time required for processing the input. We often refer to latency as *time complexity*.

• *Number of parallel streams* - the number of input streams that are processed in parallel is related to the throughput and power consumption. We say that an algorithm that has $O(B)$ parallel streams supports *scalable data rates* if its time complexity is $O(N \log(N)/B)$ or less.

There are other parameters that might be of interest, including the following.

• *Delay* - the time required for the input of a compression system to be reflected at the output.

• *Sequentiality* - if the encoder needs to process the entire input $x$ before it starts sending its output to the decoder, the delay is large.

• *Throughput* - the amount of data that the algorithm processes per time unit is proportional to the number of parallel streams.

• *Chip area* - proportional to the amount of logic and memory used.

• *Power consumption* - components of the power consumption include *dynamic power consumption*, which is proportional to the number of parallel streams and the amount of logic, and *leakage power*, which is roughly proportional to the chip’s area. In many applications, especially wireless devices that are operated by batteries, power consumption must be low.

### 4.1.3 Previous work on hardware compression and parallel compression

Previous approaches to hardware compression and parallel compression suffer from various limitations. In particular, few approaches offer a viable approach to low re-
dundancy compression that supports scalable data rates. We begin by reviewing the properties of several serial hardware compression methods; these algorithms have a single data stream, hence their data rates are limited by the clock speed. We then mention several parallel compression algorithms.

4.1.3.1 Hardware compression

Most of the work on fast lossless compression in hardware concentrated on VLSI aspects, thus outdated compression algorithms were used. For example, the implementation of LZ77 (the Lempel-Ziv string matching code [56]) has attracted much attention. Ranganathan and Henriques [57] proposed a hardware implementation of LZ77 [56] based on systolic arrays, achieving $O(N)$ time complexity instead of $O(N^2)$. Lin and Wu [58] studied low-power hardware implementations of LZ77. Both papers [57, 58] focus on hardware structures that enable linear time complexity. Unfortunately, the redundancy of LZ77 satisfies $\rho = O(N \frac{\log \log(N)}{\log(N)})$ [59], so LZ77-based methods do not achieve the redundancy bounds (1.1) and (1.2).

Another direction in hardware compression has focused on the move to front (MTF) algorithm [60]. Jones [61] proposed a hardware implementation of MTF in 1992; his data rates were impressive at that time. Myoupo and Wabbi [62] describe a hardware method for algorithms related to MTF. Jones [63] also proposed a related fixed-to-variable length code. Because the fixed-length input consisted of 32 bit words [63], a modest clock speed provided relatively high data rates. However, none of these MTF variants converge to the entropy rate, except on very specific sources.

Hardware implementation of the BWT [21] has also attracted attention. Schindler [64] considers a bounded depth BWT variant, which simplifies the BWT construction and can thus be implemented in hardware. In Chapter 5 we propose a novel suffix lists approach for implementing the BWT construction. In particular, we propose to
implement one of our algorithms in hardware. Additional authors hold patents related to hardware implementation of the BWT but, to the best of our knowledge, none of them has published their results in the academic literature. Hardware constructions of the BWT lead to nonsequential compression algorithms [3, 4, 21, 22, 25], including some low redundancy algorithms. However, these BWT constructions [64] also use a single parallel stream, hence their throughputs are limited.

4.1.3.2 Parallel compression

Stassen and Tjalkens [65] proposed a parallel compression algorithm based on context tree weighting [13] (CTW). Each symbol of the length-$N$ input is processed by one of the processors, where a single common finite state machine (FSM) determines for each symbol which processor should process it. The FSM processes the original length-$N$ input; hence it requires $O(N)$ time, that is, the processing speed is not proportional to the number of processors used. It follows that this method has limited potential for supporting scalable data rates.

Franaszek et al. [55] proposed a parallel compression algorithm, which is related to LZ77 [56], where the construction of a dictionary is divided between multiple processors. As mentioned previously, the redundancy of LZ77 is $O(N \frac{\log\log(N)}{\log(N)})$ [59]. Therefore, this parallel method does not achieve the redundancy bounds (1.1) and (1.2).

Finally, Willems [66] proposed a variant of CTW where multiple computational units process blocks of an input sequence in parallel. Partitioning the input into $B$ blocks provides $O(ND_{max}/B)$ time complexity, where $D_{max}$ is the maximal context depth that is processed. Unfortunately, the redundancy of this scheme is worse than that of CTW because (i) the first symbols at the beginning of each block appear with unknown context, and (ii) probability estimates will be based on partial information in between synchronizations of the context trees. The redundancy of this method was not analyzed by
Willems [66]. However, owing to the excellent redundancy performance of CTW [13], it seems that, among these previously proposed parallel compression methods, the parallel CTW algorithm of Willems [66] is the only method that supports scalable data rates and has relatively low redundancies.

The parallel compression method of this chapter improves on the state of the art by providing redundancy performance that is similar to that of parallel CTW [66] with a work-efficient algorithm. Furthermore, to the best of our knowledge, our analysis of the redundancy of parallel compression algorithms is novel in itself.

4.1.4 Naive parallelization

We partition the length-$N$ input $x$ into $B$ blocks of length $N/B$, where we assume that $N/B$ is an integer. A naive parallel algorithm would compress each of the $B$ blocks independently, but the redundancy would increase by approximately a factor of $B$.

Recalling the design space that was laid out in Section 4.1.2, a factor-$B$ speedup can be achieved with any compression method. However, if $N$ is fixed, increasing $B$ with naive parallelization will degrade the compression quality (numerical data that supports this claim appears in [55]). Alternatively, one could keep the block size fixed and increase $B$, hence increasing $N$, but that would increase both the system’s latency and the memory use. In the design space for parallel source coding algorithms, our goal is to provide an algorithm that, for a fixed input length $N$ and using $B$ parallel streams, has $O(N/B)$ time complexity and low redundancy. This will enable lossless compression to run at scalable data rates and may lead to a broad range of applications, which until now were limited by the throughput bottleneck of lossless serial compression algorithms.
4.1.5 Parallelizing the two-pass MDL approach

Two-pass MDL codes first describe the structure and quantized parameter estimates of the MDL source, and then encode the input $x$ based on the source structure and parameters. More specifically, two-pass MDL codes for tree sources describe $\hat{S}$ and $\hat{\Theta}$ in Phase I, and encode $x$ in Phase II. In contrast, the semipredictive approach only describes $\hat{S}$ in Phase I; the parameters $\hat{\Theta}$ are estimated sequentially in Phase II. In this chapter, we develop a parallel two-pass MDL algorithm. We do not use a semipredictive approach because estimating $B$ sets of parameters in parallel, one for encoding each of the $B$ blocks in Phase II, would generate $B$ redundancy terms, which is excessive.

We now provide a brief discussion of context trees and context tree pruning, which can be used to implement two-pass MDL algorithms [3–5, 9, 14]. Context trees store information about contexts of symbols in $x$ [5, 9, 14]. Context tree pruning (CTP) is a form of a dynamic programming algorithm for coding length minimization (c.f. Appendix A and Nohre [5] for details). In CTP, leaves of a context tree are pruned in order to obtain the MDL tree source. CTP algorithms attempt to learn the statistics of the source before the actual encoding begins; hence they can be used to implement a two-pass MDL algorithm.

In this chapter, we present a new parallel two-pass MDL method that estimates the parameters underlying the entire length-$N$ input $x$ by pruning the $B$ context trees of the $B$ blocks in parallel in Phase I, and compresses the $B$ blocks in parallel in Phase II. We call this algorithm the Parallel Two-Pass MDL (PTP-MDL) algorithm. In order to keep the presentation simple, we restrict our attention to a binary alphabet, i.e., $\mathcal{X} = \{0, 1\}$; the generalization of PTP-MDL to nonbinary alphabets is straightforward.

We first partition $x$ into $B$ blocks and construct full context trees of depth $D_{\text{max}}$ for all the blocks in parallel. Then, Phase I combines statistics from all $B$ context trees in
order to approximate the MDL tree source $S^*$ by $\hat{S}$. In Phase II, we encode each of the $B$
input blocks in parallel. In Section 4.3 we will show that a pipelined implementation of
PTP-MDL has $O(N/B)$ time complexity, while still approaching the redundancy bounds
(1.1) and (1.2). This enables scalable data rates without a factor-$B$ increase in the
redundancy.

There are several key ideas underlying this work. First and foremost, we combine
statistics from the $B$ context trees into a single parameter estimate for the entire under-
lying input $x$. Second, we avoid using $B$ KT estimators [44], one for encoding each of
the $B$ blocks in Phase II, because that would generate $B$ redundancy terms, which is ex-
cessive. Instead, we use a two-pass approach based on two-part codes [28], which encode
the conditional probability estimates explicitly in Phase I, and then use these estimates
to encode the $B$ blocks in parallel in Phase II. Third, we derive an $O(N/B)$ complexity
algorithm by restricting our attention to the class of tree sources whose maximal depth
is at most $D_{\text{max}} = \log(N/B) + O(1)$. This restriction simplifies our presentation and
allows us to incorporate several algorithms that run well for this $D_{\text{max}}$.

4.2 Basic Algorithm

4.2.1 Computational model

We assume that a length-$N$ sequence is processed on a $\log(N)$ bit computer archi-
tecture, $\log(N)$ bit words are viewed as $O(1)$ memory, and that arithmetic computa-
ions performed with up to $2\log(N)$ bits of precision are atomic operations, i.e., they require
$O(1)$ time. Our computational scheme estimates the MDL coding length within $O(1)$ bits
of the true optimum when computations are performed with $2\log(N)$ bits of precision.
This computational model is consistent with current typical 32 and 64 bit computer ar-
chitectures. Furthermore, access to $O(N)$ memory requires pointer arithmetic performed
with roughly \( \log(N) \) bits of precision, so making a distinction between “simple” operations (e.g., memory access) and “complicated” operations (e.g., divisions and logarithms) would be artificial.

Another important assumption that we make is that global control signals propagate in \( O(1) \) time. This assumption is essential, because no matter how large \( B \) is, we want control instructions to reach all computational units in \( O(1) \) time. Although this assumption is imprecise, it is common in parallel random access machine (PRAM) models [29].

4.2.2 General structure

A block diagram of a possible implementation of the PTP-MDL encoder is shown in Figure 4.1. In Phase I, the PTP-MDL encoder employs \( B \) computational units that work in parallel to accumulate statistical information on \( B \) blocks, and one unit that controls the units and computes the MDL source estimate \( \{ \hat{S}, \hat{\Theta} \} \). We call the \( B \) computational units parallel units (PUs) and call the unit that controls them the coordinating unit (CU).

Define the \( B \) blocks as \( x(1) = x_{1}^{N/B}, x(2) = x_{N/B+1}^{2N/B}, \ldots, x(B) = x_{N-N/B+1}^{N} \). Parallel unit \( b \), where \( b \in \{1, \ldots, B\} \), first computes for each depth-\( D_{\text{max}} \) context \( s \) the block
symbol counts $n_s^0(b)$ and $n_s^1(b)$, which are defined as

$$n_s^\alpha(b) = \sum_{i = D_{\text{max}} + 1}^{N/B} 1\{x_i^\alpha = s\alpha\}, \quad \alpha \in \mathcal{X}. \tag{4.1}$$

The block symbol count $n_s^\alpha(b)$ is the number of times $\alpha$ was generated by $s$ in $x(b)$.

The CU estimates the MDL source by merging statistical information from all the PUs into a single parameter estimate for the entire underlying input $x$. For each state $s$ such that $|s| < D_{\text{max}}$, the CU either retains the children states $0s$ and $1s$ in the MDL source, or prunes them and only retains $s$, whichever results in a shorter total coding length.

In Phase II, each of the $B$ blocks is compressed by a PU. For each symbol $x_i(b)$, PU $b$ first determines $G_i(b)$, the state $s$ that generated the symbol $x_i(b)$. We call $G_i(b)$ the generator of $x_i(b)$. (We determine $G_i(b)$ with the context index $c_i(b)$ and generator table $g$; see Sections 4.2.4.1 and 4.3.1.) Parallel unit $b$ then assigns $x_i(b)$ a probability according to the quantized parameters that were estimated by the CU in Phase I, and sequentially feeds the probability assignments to an arithmetic encoder [1, 10, 41]. A block diagram of Phase II appears in Figure 4.2.

The structure of the decoder is similar to that of Phase II. The approximated MDL source structure $\hat{S}$ and quantized parameters $\hat{\Theta}$ are first derived from the parallel source description (see Section 4.2.3). Then, the $B$ blocks are decompressed by $B$ decoding blocks. In decoding block $b$, each symbol $x_i(b)$ is sequentially decoded by determining

\[\text{Figure 4.2 Block diagram of Phase II.}\]
$G_i(b)$ (this is done with the context index $c_i(b)$ and generator table $g$; see Sections 4.2.4.1 and 4.3.1), assigning a probability to $x_i(b)$ based on the quantized parameter estimates, and applying an arithmetic decoder [1, 10, 41]. A block diagram of a decoding block appears in Figure 4.3. For more details on the decoder, see Section 4.2.7.

4.2.3 Parallel source description

In several semipredictive algorithms [5, 9, 14], the conditional probabilities $p(0|s)$ and $p(1|s)$, where $s \in \hat{S}$, were estimated implicitly by Krichevsky-Trofimov estimators [44] in Phase II. In PTP-MDL, using independent KT estimators in each of the PUs would generate a redundancy of approximately $\frac{B}{2} \log(N/B)$ bits per parameter, which is excessive. Instead, PTP-MDL uses two-part codes.

4.2.3.1 Two-part codes in the PTP-MDL algorithm

In Chapter 3, we proposed to use two-part codes for lossless universal source coding of i.i.d. sequences in a distributed source coding system. We adapt this approach to parallel compression of non-i.i.d. sequences by using the conditional independence of symbols given their generator state. In the first part of the code, we describe quantized estimates of the conditional probabilities explicitly. This description is generated by the CU in Phase I. In the second part of the code, the sequence is encoded based on the parameter estimates. This encoding is performed in parallel in Phase II.
The CU first determines the symbol counts

\[ n_s^\alpha = \sum_{b=1}^{B} n_s^0 (b), \quad \alpha \in \mathcal{X}. \]  

(4.2)

The symbol counts \( n_s^0 \) and \( n_s^1 \) are the number of times 0 and 1 were generated by state \( s \) in the entire sequence \( x \), i.e., in any of the \( B \) blocks. The CU can then compute the ML parameter estimates of \( p(1|s) \) and \( p(0|s) \),

\[ \theta_s^1 = \frac{n_s^1}{n_s^0 + n_s^1} \quad \text{and} \quad \theta_s^0 = 1 - \theta_s^1, \]

respectively. For brevity, we also define the ML parameter as \( \theta_s \triangleq \theta_s^1 \). The ML parameter passes through a bijective mapping\(^1\)

\[ \eta_s \triangleq \frac{2}{\pi} \arcsin \left( \sqrt{\theta_s} \right), \]

where \( \eta_s \in [0, 1] \). The bin index \( k_s \) is determined by uniformly quantizing \( \eta_s \), so

\[ k_s \triangleq \begin{cases} 
[\eta_s K_s] & \eta_s \in (0, 1] \\
1 & \eta_s = 0
\end{cases}, \]

where \( [\cdot] \) denotes rounding up and \( K_s \), which will be specified in Section 4.2.3.3, is the number of quantization bins for state \( s \). Note that in bin \( k_s \) we have \( \eta_s \in \left[ \frac{k_s - 1}{K_s}, \frac{k_s}{K_s} \right] \) and \( \theta_s \in \left[ \sin^2 \left( \frac{\pi (k_s - 1)}{2K_s} \right), \sin^2 \left( \frac{\pi k_s}{2K_s} \right) \right] \).

All values of \( \theta_s \) in bin \( k_s \) are quantized to a single representation level \( r_s \). We choose \( r_s \) to lie in the middle of the bin in \( \eta(\cdot) \), i.e., \( \frac{2k_s - 1}{2K_s} \), which corresponds to

\[ r_s \triangleq \sin^2 \left( \frac{\pi (2k_s - 1)}{4K_s} \right). \]

Denoting the quantized ML estimate of \( \theta_s^\alpha \) by \( \hat{\theta}_s^\alpha \), we have \( \hat{\theta}_s^1 = r_s \) and \( \hat{\theta}_s^0 = 1 - r_s \). At the end of Phase I, the CU has computed the MDL structure estimate \( \hat{S} \). If \( s \in \hat{S} \), the

---

\(^1\)This mapping is motivated by the observation that for \( \theta_a, \theta_b, \theta_c, \) and \( \theta_d \) satisfying \( \eta(\theta_b) - \eta(\theta_a) = \eta(\theta_d) - \eta(\theta_c) \), we have \( D(\theta_a||\theta_b) \approx D(\theta_c||\theta_d) \), where \( D(\cdot||\cdot) \) are Kullback-Leibler divergences (see Section 3.4 and references therein).
first part of the two-part code for symbols generated by $s$ consists of encoding $k_s$ with 
log($K_s$) bits.

In Phase II, which implements the second part of the two-part code, each PU $b$ encodes its block $x(b)$ sequentially. For each symbol $x_i(b)$, PU $b$ determines $G_i(b)$, the 
generator of $x_i(b)$. The symbol $x_i(b)$ is encoded according to the probability assignment 
$\hat{p}(x_i(b)) \triangleq \hat{\theta}^{x_i(b)}_{G_i(b)}$ with an arithmetic encoder [1, 10, 41]. The probability assigned by PU $b$
to the symbols in $x(b)$ whose generator is $s$ is

$$
\prod_{\{i: G_i(b)=s, i>D_{max}\}} \hat{p}(x_i(b)) = (r_s)^{n_s^1(b)}(1 - r_s)^{n_s^0(b)}.
$$

(4.3)

The product of the probabilities assigned by all $B$ PUs to symbols whose generator is $s$
is thus

$$
\prod_{b=1}^{B} \prod_{\{i: G_i(b)=s, i>D_{max}\}} \hat{p}(x_i(b)) = (r_s)^{n_s^1}(1 - r_s)^{n_s^0}.
$$

(4.4)

We also define $n_s \triangleq n_s^0 + n_s^1$. In (4.4), the quantized parameter estimate $\hat{\theta}_s = r_s$ is used
in the second part of the code for all the symbols whose generator is $s$. This property
(4.4) provides the same redundancy for two-part codes in a distributed system as we
would obtain in a nondistributed system (see Section 3.3.2).

4.2.3.2 Worst-case redundancy of two-part codes

A two-part code for i.i.d. sequences over a binary alphabet was outlined by Stine and Fost
er [53]. That code is based on Jeffreys’ prior [28, 51, 53]. Using the optimal number of bins, which will be specified in Section 4.2.3.3, we obtain what we call the Jeffreys code. In Chapter 3, we analyzed the worst-case redundancy (WCR) of the Jeffreys code for i.i.d. sequences, and optimized the number of bins. Theorem 8 upper bounded the
WCR of the Jeffreys code for a length-$N$ i.i.d. input sequence over a binary alphabet by
$\frac{1}{2} \log(N) + 1.547 + O(\frac{1}{N})$ bits, which is $1.221 + O(\frac{1}{N})$ bits above Rissanen’s bound.
In our PTP-MDL algorithm, we adapt the Jeffreys code to parallel compression of non-i.i.d. sequences by using the conditional independence of symbols given their generator state. Using (4.4), the product of the probabilities assigned by all $B$ PUs to the symbols $x_{D_{max} + 1}^{N/B}(b)$ is

$$\prod_{b=1}^{B} \prod_{i=D_{max}+1}^{N/B} \hat{p}(x_i(b)) = \prod_{b=1}^{B} \prod_{i=D_{max}+1}^{N/B} \left[ l_{x_i(b)=1} \cdot r_{G_i(b)} + l_{x_i(b)=0} \cdot (1 - r_{G_i(b)}) \right]$$

$$= \prod_{s \in \hat{S}} \left( r_s \right)^{n_s} (1 - r_s)^{n_0}$$

$$= \prod_{s \in \hat{S}} \prod_{b=1}^{B} \prod_{i: G_i(b) = s, i > D_{max}} \hat{p}(x_i(b)).$$

Therefore, the symbols generated by each state $s$ are compressed by the Jeffreys code for that state just like an i.i.d. sequence over a binary alphabet.

Two-part codes can also be used for nonbinary alphabets (see Chapter 3). An extension of the Jeffreys code yields a WCR of $(|X| - 1) [1.221 + o(1)]$ bits above Rissanen's bound. Furthermore, we proved that two-part codes with a modified quantization structure can attain a WCR as low as $(|X| - 1) [1.047 + o(1)]$ bits above Rissanen's bound. However, the WCR of the Jeffreys code is sufficiently low for this chapter.

### 4.2.3.3 Number of quantization bins

We now specify $K_s$, the number of bins used for quantizing the ML parameter $\theta_s$. In Section 3.4.3, we showed that $K^*$, the optimal number of bins for encoding length-$N$ sequences, satisfies

$$K^* = \sqrt{2\pi^2 \ln(2) \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) N + O(1) \approx 1.772 \sqrt{N}.} \quad (4.5)$$

In most of this chapter, we use

$$K_s \triangleq \left\lfloor \sqrt{2\pi^2 \ln(2) \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) N} \right\rfloor. \quad (4.6)$$
However, if $n_s < N$, i.e., there are less than $N$ symbols whose generator is $s$, Theorem 8 implies that we could save $\frac{1}{2} \log(N/n_s)$ bits by using a two-part code that is optimal for length-$n_s$ sequences. Accordingly, in Section 4.5.1 we will derive parallel source description methods that describe coarse estimates of $n_s$ in order to decrease the coding length by using two-part codes with fewer bins. In any case, if $n_s < N$, the redundancy with $K_s$ given in (4.6) is less than $\frac{1}{2} \log(N) + 1.547 + O(1/N)$ bits (see Section 4.2.3.2), so the WCR obtained is sufficiently low for this chapter.

4.2.3.4 Source description in Phase I and Phase II

We now specify the parallel source description used by PTP-MDL. The coding length using this description will be minimized in Section 4.2.4.2. In Phase I, the CU describes the approximated MDL tree source structure $\hat{S}$ and its quantized ML parameters $\hat{\Theta}$. The structure $\hat{S}$ is described with the natural code [5, 13]. The code natural$_s$ of a subtree rooted at $s \in X^*$ is empty if $|s| = D_{\text{max}}$; otherwise, it is 0 if $s \in \mathcal{S}$, else it is 1 followed by natural$_{0s}$ and natural$_{1s}$. We describe $\hat{S}$ explicitly by the code natural$_\lambda$ of the entire tree rooted at the empty sequence $\lambda$. Alternatively, this code is denoted by natural$_{\hat{S}}$. For a binary alphabet, $|\text{natural}_{\hat{S}}| \leq 2|\hat{S}| - 1$ bits; this is the model redundancy of PTP-MDL.

In addition to natural$_{\hat{S}}$, in Phase I the CU describes the $|\hat{S}|$ indices $k_s$ in the order in which the leaves of $\hat{S}$ are reached in a depth-first search [29]; this description can be implemented with arithmetic coding [1, 10, 41]. The corresponding coding length is the parameter redundancy of PTP-MDL. We denote the length of the description of $\hat{S}$ and the $|\hat{S}|$ bin indices generated in Phase I by $l_s^I$. Using (4.5),

$$l_s^I = |\text{natural}_{\hat{S}}| + |\hat{S}| \log(K_s)$$

$$\leq 2|\hat{S}| - 1 + |\hat{S}| \left[ \log(1.772) + \frac{1}{2} \log(N) \right]$$

$$= |\hat{S}| \left[ 2.825 + \frac{1}{2} \log(N) \right] - 1 \text{ bits.}$$

105
In Phase II, each PU \( b \) determines \( G_i(b) \) for all the symbols of \( x(b) \), and assigns those symbols probabilities according to (4.3). That is the main component of the code generated by the PU \( b \) in Phase II. There are two additional terms that affect the coding length in Phase II.

- **Coding redundancy** - each arithmetic encoder may add coding redundancy [1]. Arithmetic codes implemented with finite precision may increase the coding length further [10,41]. However, if arithmetic coding is performed with \( \log(N) \) bits of precision, the expected coding redundancy is \( O(1) \) bits [10]. Because the amount of coding redundancy can only be determined after encoding the sequence, we neglect the coding redundancy in the coding length minimization of Phase I, and only take it into account when discussing the total redundancy in Section 4.4.2.

- **Symbols with unknown context** - at the beginning of \( x(b) \), there may be symbols \( x_1(b) \) whose context \( x_1(b) \) is not deep enough to determine \( G_i(b) \). We call these *symbols with unknown context*. In order to avoid the determination of \( G_i(b) \) for symbols with unknown context, we encode the first \( D_{max} \) symbols of each block \( x(b) \) directly, which requires \( D_{max} \) bits per block. Note that the definition (4.1) of \( n_s^0(b) \) omits these symbols with unknown contexts by counting symbols in \( x_{D_{max}+1}^N(b) \), not \( x_1^{N/B}(b) \).

Neglecting the coding redundancy, we denote the length of the code generated by PU \( b \) in Phase II by \( l_{s}^{II}(b) \). Taking symbols with unknown context into account,

\[
l_{s}^{II}(b) = D_{max} - \sum_{s \in \mathcal{S}} [n_s^1(b) \log(r_s) + n_s^0(b) \log(1 - r_s)]
\]

bits. Denoting the combined length of all \( B \) codes in Phase II by \( l_{s}^{II} \), we have

\[
l_{s}^{II} = B \cdot D_{max} - \sum_{s \in \mathcal{S}} [n_s^1 \log(r_s) + n_s^0 \log(1 - r_s)] \tag{4.7}
\]

106
bits. We can think of $2^{-i^{\|}}$ as the probability of the input sequence $x$ conditioned on the
tree source structure $\mathcal{S}$ and its quantized ML parameters, which is analogous to $p_\mathcal{S}(x)$ of
the semipredictive approach. Because each symbol $x_i(b)$ is assigned a probability $r_{G_i}(b)$
or $1 - r_{G_i}(b)$ (with $r_{G_i}(b) = 0.5$ for $i \leq D_{\text{max}}$), this parallel source description method
 corresponds to a valid probability mass function.

4.2.3.5 MDL tree source

Now that the source description method has been defined, we define an MDL tree
source $\{\mathcal{S}^*, \Theta^*\}$ as a tree source whose explicit description with the natural code and
quantized parameters, along with the coding length of Phase II, provides the shortest
description of the data. Therefore,

$$\mathcal{S}^* \triangleq \arg \min_{\mathcal{S} \in \mathcal{D}_{\text{max}}} \{l^I_{\mathcal{S}} + l^{II}_{\mathcal{S}}\},$$

where $\mathcal{D}_{\text{max}}$ is the class of tree source structures whose maximal depth is at most $D_{\text{max}}$.

In Section 4.2.4.2 we will explain that $\hat{\mathcal{S}}$, the outcome of Phase I, is an MDL tree source
structure. We thus refer to $\hat{\mathcal{S}}$ as an MDL tree source structure.

4.2.4 Phase I

4.2.4.1 Constructing context trees (computing symbol counts)

Before the CU can gather statistical information from all $B$ context trees in parallel,
the PUs must construct the context trees [5, 9, 14]. Because we restrict our attention to
depth-$D_{\text{max}}$ contexts, it suffices for PU $b$ to compute $\{n^a_\xi(b)\}_{a \in \mathcal{X}}$, all the block
symbol counts of all the leaf contexts of a full depth-$D_{\text{max}}$ context tree. These block
symbol counts contain all the relevant (depth-$D_{\text{max}}$) information that a context tree for
$x(b)$ contains. Furthermore, as we show, information on internal nodes of the context
tree, whose depth is less than $D_{\text{max}}$, can be computed from the block symbol counts of
the leaf contexts.
For each state $s$ that is processed, the CU needs to determine the coding length required for retaining $s$ in $\mathcal{S}$. The coding length is computed using $n_s^0$ and $n_s^1$. If $|s| = D_{\text{max}}$, the CU gets $\{n_s^a(b)\}_{a \in \mathcal{X}, b \in \{1, \ldots, B\}}$ from the PUs and computes $n_s^0$ and $n_s^1$ with (4.2). Alternatively, $|s| < D_{\text{max}}$, and the CU recursively derives $n_s^0$ and $n_s^1$ by adding up the symbol counts of children states, i.e.,

$$n_s^a = n_{os}^a + n_{is}^a, \quad \forall \alpha \in \mathcal{X}. \quad (4.8)$$

Computational unit $b$ computes $n_s^a(b)$ for all $2^{D_{\text{max}}}$ depth-$D_{\text{max}}$ leaf contexts $s$ by constructing an array that stores all the block symbol counts. Parallel unit $b$ then scans $x(b)$ sequentially and increments the relevant block symbol count for each $x_i(b)$. The trivial implementation of this procedure would determine the leaf context of $x_i(b)$ according to the previous $D_{\text{max}}(N)$ symbols. However, because $D_{\text{max}}(N) \approx \log(N/B)$, this approach has $O(N/B \log(N/B))$ complexity, which is excessive.

In order for PU $b$ to compute all block symbol counts in $O(N/B)$ time, we define the context index $c_i(b)$ of the symbol $x_i(b)$ as

$$c_i(b) \triangleq \sum_{j=0}^{D_{\text{max}}-1} 2^j x_{j+i-D_{\text{max}}}(b), \quad (4.9)$$

where $i \in \{D_{\text{max}} + 1, \ldots, N/B\}$ and $x_{j+i-D_{\text{max}}}(b) \in \{0, 1\}$, hence $c_i(b) \in \{0, \ldots, 2^{D_{\text{max}}-1}\}$. Note that $c_i(b)$ is the binary number represented by the context $s = x_{i-D_{\text{max}}}(b)$. Hence, it can be used as a pointer to the address containing the block symbol count $n_s^a(b)$ for $s = x_{i-D_{\text{max}}}(b)$. We thus define the block symbol counts array $f$ such that

$$f_{2^{D_{\text{max}}-1}x_{i}(b)+c_{i}(b)} = n_{s}^a(x_{i}(b)), \quad \text{by incrementing the cells of the array by one every time the context addresses them.}$$

We have the useful property

$$c_{i+1}(b) = \frac{c_i(b)}{2} + 2^{D_{\text{max}}-1}x_i(b) - \frac{x_{i-D_{\text{max}}}(b)}{2}, \quad (4.10)$$

108
1. in parallel for $b \in \{1, \ldots, B\}$ 
   
2. for $i = 0$ to $2^{D_{\text{max}}} + 1 - 1$ /* initialize $f$ */ 
   
3. $f_i \leftarrow 0$ 
   
4. } 
   
5. Compute $c_{D_{\text{max}}+1}(b)$ using (4.9) 
6. for $i = D_{\text{max}} + 1$ to $N/B$ 
   
7. $f_{2D_{\text{max}}x_i(b)+c_i(b)} \leftarrow f_{2D_{\text{max}}x_i(b)+c_i(b)} + 1$ /* $f_{2D_{\text{max}}x_i(b)+c_i(b)} = n_{x_i^{-1}_{D_{\text{max}}}}^{(b)}(b)$ */ 
   
8. Compute $c_{i+1}(b)$ using (4.10) 
9. } 
10. } 

**Figure 4.4** Pseudocode for computing the block symbol counts of all leaf contexts.

which implies that our computational model enables computing all $N/B - D_{\text{max}}$ context indices of the symbols of $x(b)$ in $O(N/B)$ time complexity. Pseudocode for computing the block symbol counts of all leaf contexts appears in Figure 4.4. The in parallel structure in Line 1 encloses a segment of code that is executed on all $B$ PUs at the same time. Initializing $f$ in Lines 2-4 requires $O(N/B)$ time complexity because $f$ has $2^{D_{\text{max}}+1} = O(N/B)$ entries. The aggregate complexity of Lines 6-9 is also $O(N/B)$ because each symbol is processed in $O(1)$ time. With this algorithm, the PUs compute the block symbol counts, hence constructing the context trees, in $O(N/B)$ time.

### 4.2.4.2 Computing the MDL length

For each state $s$, we either retain the children states 0s and 1s in the tree or merge them into a single state, according to which decision minimizes the coding length. The coding length $l_s$ of the two-part code that describes the symbols generated by $s$ is

$$l_s = \underbrace{\log(K_s)}_{\text{Part I}} - \underbrace{n_s^0 \log(1 - r_s)}_{\text{Part II}} - \underbrace{n_s^1 \log(r_s)}_{\text{Part II}}.$$  

(4.11)
For nonbinary alphabets, a similar expression can be derived (see Section 3.6).

We now derive the coding length required for state \( s \), which is denoted by \( \text{MDL}_s \).
For \( |s| = D_{\text{max}} \), \( n_s^0 \) and \( n_s^1 \) are computed with (4.2), \( l_s \) is computed with (4.11), and \( \text{MDL}_s = l_s \). For \( |s| < D_{\text{max}} \), we compute \( \text{MDL}_s \) hierarchically with (4.8), after already having processed the children states. In order to decide whether to prune the tree, we compare \( \text{MDL}_{0s} + \text{MDL}_{1s} \) with \( l_s \). Because retaining an internal node requires the natural code [5, 13] to describe that node (with 1 bit),

\[
\text{MDL}_s = \begin{cases} 
  l_s & \text{if } |s| = D_{\text{max}}, \\
  1 + \min \{ \text{MDL}_{0s} + \text{MDL}_{1s}, \ l_s \} & \text{otherwise}
\end{cases}
\]

In terms of the natural code, if \( |s| = D_{\text{max}} \), \( s \) is a leaf of the full depth-\( D_{\text{max}} \) context tree, and its natural code is empty, else \( |s| < D_{\text{max}} \), and the natural code requires 1 bit to encode whether \( s \in \mathcal{S} \). The symbols generated by \( s \) are either encoded by retaining the children states (this requires a coding length of \( \text{MDL}_{0s} + \text{MDL}_{1s} \) bits), or pruning the children states and retaining state \( s \) with coding length \( l_s \). If \( |s| = D_{\text{max}} \), we do not process deeper contexts, hence the coding length is always \( l_s \).

Phase I in PTP-MDL determines the MDL tree source structure \( \hat{\mathcal{S}} \), the resulting coding length, and the natural code for \( \hat{\mathcal{S}} \), by computing \( \text{MDL}_\lambda \), where \( \lambda \) is the empty sequence. The outcome of Phase I is an MDL structure because CTP can be shown to be a form of a dynamic programming algorithm for coding length minimization (c.f. Appendix A and Nohre [5] for details).

Pseudocode for the \( \text{PMDL} \) routine appears in Figure 4.5. The routine recursively processes all context tree nodes whose depth is up to \( D_{\text{max}} \). The \( \text{PMDL} \) routine outputs \( \text{MDL}_s \), \( n_s^0 \), \( n_s^1 \), and \( \text{natural}_s \), the natural code for the MDL subtree that best describes the symbols generated by state \( s \). Phase I runs \( \text{PMDL}(\lambda) \), which outputs \( \text{MDL}_\lambda \) and \( \text{natural}_\lambda = \text{natural}_\hat{\mathcal{S}} \). In view of (4.7), encoding \( x \) requires an additional \( B \cdot D_{\text{max}} \) bits for encoding the first \( D_{\text{max}} \) symbols of each block directly, but the \( \text{PMDL} \) routine does
1. **routine** $PMDL(s)$
2. if ($|s| = D_{max}$) { /* $s$ is a leaf */
3. in parallel for $b \in \{1, \ldots, B\}$ {
4. Send $n^0_s(b)$ and $n^1_s(b)$ from PU $b$ to the CU
5. }
6. Compute $n^0_s$ and $n^1_s$ using (4.2)
7. Compute $l_s$ using (4.11)
8. $MDL_s \leftarrow l_s$
9. $natural_s \leftarrow \lambda$
10. } else { /* $s$ is an internal node */
11. Compute $MDL_{0s}$, $n^0_{0s}$, $n^1_{0s}$, and $natural_{0s}$ by running $PMDL(0s)$
12. Compute $MDL_{1s}$, $n^0_{1s}$, $n^1_{1s}$, and $natural_{1s}$ by running $PMDL(1s)$
13. Compute $n^0_s$ and $n^1_s$ using (4.8)
14. Compute $l_s$ using (4.11)
15. if ($l_s > MDL_{0s} + MDL_{1s}$) { /* retain children states */
16. $MDL_s \leftarrow 1 + MDL_{0s} + MDL_{1s}$
17. $natural_s \leftarrow 1 natural_{0s} natural_{1s}$
18. } else { /* prune the tree */
19. $MDL_s \leftarrow 1 + l_s$
20. $natural_s \leftarrow 0$
21. }
22. } return $MDL_s$, $n^0_s$, $n^1_s$, and $natural_s$

**Figure 4.5** Pseudocode for the $PMDL$ routine.

not account for this. (The arithmetic coding redundancy is ignored, as stated earlier.)

An example of the execution of the $PMDL$ routine appears in Appendix K.

### 4.2.4.3 Computing the MDL length with finite precision

The computation of the MDL coding length requires adding up, per (4.7), $O(|\hat{S}|)$ coding length terms of the form $n^0_s\log(\hat{\theta}^0_s)$. However, these calculations are performed at a finite numerical precision. We now explain why our computational model enables the estimation of the MDL coding length within $O(1)$ bits of the true optimum.
In order to attain the required accuracy, the aggregate error over all $O(|\tilde{S}|)$ coding length terms (4.7) must be $O(1)$ at most. Because $|\tilde{S}| \leq 2^{D_{\max}}$, there may be $O(2^{D_{\max}+1}) = O(N/B)$ terms; hence, we must compute $n_s^o \log(\hat{\theta}_s^o)$ to an absolute precision of $O(B/N)$. But $n_s^o$ may be as large as $O(N)$, so $\log(\hat{\theta}_s^o)$ must be computed to an absolute precision of $O(B/N^2)$, which requires performing the computation of $\log(\hat{\theta}_s^o)$ with $\log(N^2/B) + O(1)$ bits of precision. Recall from Section 4.2.1 that, with our computational model, computations performed with $2\log(N)$ bits of precision require $O(1)$ time. Therefore, using our computational model, and neglecting the coding redundancy, a finite precision implementation of Phase I approximates the MDL coding length within $O(1)$ bits of the true optimum.

**Theorem 15** When computations are performed with $2\log(N)$ bits of precision, Phase I approximates the MDL coding length within $O(1)$ bits of the true optimum.

We now consider the complexity of the *PMDL* routine (Figure 4.5). Each function call to the routine processes a state of the bounded depth context tree, so there are $O(2^{D_{\max}}) = O(N/B)$ function calls. For each function call, Line 6 requires $O(B)$ time, and the rest of the routine requires $O(1)$ time. Hence, the *PMDL* routine of Figure 4.5 needs $O(N)$ time. In Section 4.3, we will reduce the complexity of the *PMDL* routine, and ultimately Phase I, to $O(N/B)$.

**4.2.5 Phase II**

In Section 4.2.3.1, we discussed how in Phase I the PTP-MDL algorithm describes $\hat{\theta} = r_s$, the quantized ML parameter estimate of the conditional probability $p(1|s)$. In Phase II, PU $b$ knows $\tilde{S}$ and $\{r_s\}_{s \in \tilde{S}}$. PU $b$ encodes $x(b)$ sequentially; for each symbol $x_i(b)$, it determines $G_i(b)$. An $O(N/B)$ algorithm for determining $G_i(b)$ for all the symbols of $x(b)$ is described in Section 4.3.1. After determining $G_i(b)$, the symbol $x_i(b)$ is encoded
according to the probability assignment \( \hat{p}(x_i(b)) \triangleq \hat{\theta}_{x_i(b)}^{c_i(b)} \) with an arithmetic encoder [1, 10, 41]. In order to have \( O(1) \) expected coding redundancy per PU, arithmetic coding is performed with \( \log(N) \) bits of precision [10]. With our computational model, arithmetic coding requires \( O(N/B) \) complexity.

### 4.2.6 Encoder output stage

Recall that we denote the length of the description of the MDL source structure and quantized parameters produced by Phase I by \( l^I_{\hat{S}} \), and the length of the compressed block generated by PU \( b \) in Phase II by \( \hat{l}^H_{\hat{S}}(b) \) (in both terms, the arithmetic coding redundancy is neglected). Including arithmetic coding redundancy, we denote the output length of the source description of Phase I, the output length of the compressed block generated by PU \( b \), and the total output coding length by \( \hat{l}^I_{\hat{S}}, \hat{l}^H_{\hat{S}}(b) \) and \( l_{\text{total}} \), respectively. Note that \( \hat{l}^I_{\hat{S}}, \hat{l}^H_{\hat{S}}(b) \in \mathbb{N} \) and that we have

\[
l_{\text{total}} = \hat{l}^I_{\hat{S}} + \sum_{b=1}^{B} \hat{l}^H_{\hat{S}}(b) \in \mathbb{N}.
\]

We describe two options for the encoder’s output interface that illustrate different trade-offs between design simplicity and redundancy.

One option is to interleave the source description of Phase I and the \( B \) compressed blocks onto a \( B' \) bit bus, where we assume \( B' = o(B) \). We first output the source description of Phase I because the decoder must know the structure and quantized parameters \( \{\hat{S}, \hat{\Theta}\} \) in order to operate properly. After the source description of Phase I, we interleave the \( B \) compressed blocks onto \( B' \) lines. Define

\[
\hat{l}^H_{\text{min}} \triangleq \min_{b \in \{1, \ldots, B\}} \hat{l}^H_{\hat{S}}(b).
\]

The first \( \hat{l}^H_{\text{min}} \) bits of all \( B \) compressed blocks are interleaved onto \( B' \) lines. Once some of the compressed blocks are finished, the remaining compressed blocks are interleaved
onto $B'$ lines. Owing to the structure and properties of arithmetic coding [1, 10, 41],
the decoder (see Section 4.2.7) knows when a compressed block is finished; hence, the
decoder can synchronize its demultiplexing mechanism to the incoming bitstream. The
bus contains chunks of $B'$ bits, so there are $[l_{\text{total}}/B']$ chunks. The last chunk may contain
less than $B'$ (meaningful) bits; we call the portion of the last chunk that is meaningless
implementation redundancy. The implementation redundancy is less than $B' = o(B)$ bits,
which is negligible. Finally, interleaving $B$ compressed blocks may require buffering in
between the PUs that generate the compressed blocks and the multiplexer that generates
the $B'$ bit bus. A block diagram of an interleaving scheme appears in Figure 4.6.

Another option is to map the $B$ compressed blocks directly onto $B$ output lines (the
parameter description of Phase I may be transmitted on any of the $B$ lines or on a
separate line). This scheme requires $\overline{l}_{\text{max}}$ chunks of $B$ bits, where

$$\overline{l}_{\text{max}} \triangleq \max_{b \in \{1, \ldots, B\}} \overline{l}_{\mathcal{S}}(b).$$

Unfortunately, if the compressed blocks vary greatly in length, this scheme has a much
higher implementation redundancy than the interleaving scheme because (i) larger chunks

Figure 4.6 Block diagram of an interleaving scheme in the encoder.
are used, and (ii) many chunks may not be full. We leave the analysis of the implementation redundancy of this scheme for future research.

### 4.2.7 Decoder

Assuming an interleaving scheme in the encoder, the decoder receives an incoming bitstream on a $B'$-bit bus. The stream begins with the source description of Phase I, and then contains the descriptions of the $B$ compressed blocks. The decoder demultiplexes the bus and reconstructs $\mathcal{S}$ and $\{r_s\}_{s \in \hat{S}}$, which are forwarded to $B$ decoding blocks. The $B$ compressed blocks are then demultiplexed and decompressed. Finally, we need to transmit the reconstructed blocks one after another. This may require buffering and multiplexing to a serial output. A block diagram of the decoder appears in Figure 4.7.

The $B$ decoding blocks can be implemented on $B$ PUs. Decoding block $b$ decodes $x(b)$ sequentially; for each symbol $x_i(b)$, it determines $G_i(b)$. The same $O(N/B)$ algorithm used in Phase II for determining $G_i(b)$ for all the symbols of $x(b)$ can be used in the $B$ decoding blocks (see Section 4.3.1). After determining $G_i(b)$, the symbol $x_i(b)$ is decoded according to the probability assignment $\hat{p}(x_i(b)) = \hat{\theta}_{G_i(b)}$ with an arithmetic decoder.
[1, 10, 41]. Arithmetic decoding is performed with \( \log(N) \) bits of precision [10]. With our computational model, arithmetic decoding requires \( O(N/B) \) complexity. A block diagram of a decoding block is shown in Figure 4.3.

### 4.3 An \( O(N/B) \) Complexity Algorithm

In Sections 4.2.5 and 4.2.7, we pointed out that the decoder and Phase II both need to determine \( G_i(b) \) for each symbol \( x_i(b) \). Section 4.3.1 explains how we do this efficiently, and thus achieve \( O(N/B) \) complexity in Phase II and in the decoder. In order to also attain \( O(N/B) \) complexity in Phase I, we need to compute \( \{n^a_s\}_{a \in \mathcal{X}_s}, s \in \mathcal{X}^{D_{max}} \) in \( O(N/B) \) time, whereas the PMDL algorithm requires \( O(N) \) aggregate time for this computation in Line 6 (see the discussion at the end of Section 4.2.4.3). Section 4.3.2 explains how we can compute \( n^0_s \) and \( n^1_s \) in \( O(\log(B)) \) time per state \( s \). Section 4.3.3 shows how pipelining can further reduce the complexity of the PMDL routine, and ultimately Phase I, to \( O(N/B) \).

#### 4.3.1 Determining the generators of all the symbols

In the decoder and in Phase II, \( \hat{S} \) is known. Each PU \( b \) processes \( x(b) \) sequentially; for each symbol \( x_i(b) \), PU \( b \) needs to determine \( G_i(b) \). We provide an \( O(N/B) \) algorithm that determines \( G_i(b) \) for all the symbols of \( x(b) \). The algorithm constructs a length-\( 2^{D_{max}} \) generator table \( g \) such that \( g_j = s \) if \( s \) is a suffix of the length-\( D_{max} \) context whose context index is \( j \). The generator table will enable to compute the generators of all the symbols of \( x(b) \) in \( O(N/B) \) time complexity because \( G_i(b) = g_{c_i(b)}, \) and all \( N/B - D_{max} \) context indices of the symbols of \( x(b) \) can be computed in \( O(N/B) \) time complexity (see Section 4.2.4.1). Nohre [5] used a similar concept for serial compression, wherein the context index is used to access the generator table \( g \), which returns the generator \( G_i(b) \). However, Nohre did not explain how to construct \( g \).
The construction of \( g \) from \( \hat{S} \) relies on the following observations. If \( s \) is a suffix of \( x_{i-D_{max}}^{i-1} \), then \( s = s_1^{i-1} = x_{i-1}^{i-1} \), and using (4.9) we have

\[
c_i(b) = \psi_i(b) + \sum_{j=0}^{D_{max}-|s|-1} 2^j x_{j+i-D_{max}}(b) + \sum_{j=D_{max}-|s|}^{D_{max}-1} 2^j x_{j+i-D_{max}}(b)
\]

\[
= \psi_i(b) + \sum_{k=1}^{|s|} 2^{D_{max}-k} x_{i-k}(b)
\]

\[= \psi_i(b) + \left[ \sum_{k=1}^{|s|} 2^{D_{max}-k} s_{i+1-k} \right],\]

where \( \psi_i(b) \in \{0, \ldots, 2^{D_{max}-|s|} - 1\} \) and we substituted \( k = D_{max} - j \). Observe now that \( g_j = s \) for \( j \in \{\psi_s, \ldots, \psi_s + 2^{D_{max}-|s|} - 1\} \), and \( \psi_s \) are ordered according to the lexicographic order of all the states of \( \hat{S} \), where the last symbol of a suffix is most significant. Furthermore, because \( \hat{S} \) is complete, the Kraft inequality [1] implies that \( \sum_{s \in S} 2^{D_{max}-|s|} = 2^{D_{max}} \). Therefore, the state \( s_{small} \) that is smallest lexicographically satisfies \( \psi_{s_{small}} = 0 \), and the state \( s_{large} \) that is largest lexicographically satisfies \( \psi_{s_{large}} + 2^{D_{max}-|s_{large}|} = 2^{D_{max}} \).

Following these observations, we process the states of \( \hat{S} \) in increasing lexicographic order by recursively processing its representation \( natural_{S} \). We begin with the state \( s_{small} \). For each state \( s \), we assign the value \( s \) to \( 2^{D_{max}-|s|} \) locations in the generator table \( g \), and determine \( \psi_{s'} = \psi_s + 2^{D_{max}-|s|} \), where \( s' \) is the state that comes after \( s \) in lexicographic order. Pseudocode for the \textit{scan} routine appears in Figure 4.8; we invoke it with \( s = \lambda \) and \( \psi = 0 \). The routine \textit{scan} processes the states of \( \hat{S} \) in increasing lexicographic order by recursively processing 0s and 1s in Lines 7 and 8, respectively. The function call \textit{getbit} in Line 2 returns the next bit in the natural code \( natural_{S} \). The complexity of running \textit{scan}(\( \lambda, 0 \)) is \( O(2^{D_{max}}) = O(N/B) \) because the loop (Lines 3-5) requires \( O(2^{D_{max}}) = O(N/B) \) aggregate time, the rest of the routine requires \( O(1) \) time.
1. routine \textit{scan}(s, \psi)

2. if (getbit(natural\_s) = 0 or |s| = D\_\textit{max}) \{ /* s is a leaf */

3. for \( j = \psi \) to \( \psi + 2^{D\_\textit{max}} - |s| - 1 \) \{

4. \( g_j \leftarrow s \)

5. \} \( \psi \leftarrow \psi + 2^{D\_\textit{max}} - |s| \)

6. \} else \{ /* s is an internal node */

7. \( \psi \leftarrow \text{scan}(0s, \psi) \)

8. \( \psi \leftarrow \text{scan}(1s, \psi) \)

9. \} return \( \psi \)

\textbf{Figure 4.8} Pseudocode for constructing the generator table \( g \), where \( g_j = s \) if \( s \) is a suffix of the length-\( D\_\textit{max} \) context whose context index is \( j \).

per function call, and there are \( O(|\tilde{S}|) \leq O(N/B) \) function calls. Hence \( g \) is constructed from \textit{natural}_s in \( O(N/B) \) time, leading to the determination of the generators of all the symbols of \( x(b) \) in \( O(N/B) \) time complexity.

The construction of the generator table \( g \) for \( x(b) \) is used in both Phase II of the encoder and the decoder. Pseudocode for Phase II appears in Figure 4.9. Because the main loop (Lines 6-9 in Figure 4.9) requires \( O(1) \) time per symbol, Phase II has \( O(N/B) \) complexity. The decoder is also \( O(N/B) \); the main difference is that the decoder uses arithmetic decoding in Lines 4 and 7 instead of encoding.

\subsection{4.3.2 Adding up block symbol counts}

Line 6 of the \textit{PMDL} pseudocode in Figure 4.5 computes \( n^0_s \) and \( n^1_s \) using (4.2), which requires \( B \) additions for each \( n^0_s \) value. Whereas a sequential implementation requires \( O(B) \) time, an adder tree in VLSI has \( O(\log(B)) \) time complexity (Figure 4.10). However, even this speedup is insufficient by itself. Because Phase I computes symbol counts for
1. **in parallel** for $b \in \{1, \ldots, B\}$ {
2. Construct $g$ by running $\text{scan}(\lambda, 0)$ /* see Figure 4.8 */
3. **for** $i = 1$ to $D_{\text{max}}$ { /* symbols with unknown context */
4. Encode $x_i(b)$ directly
5. } Compute $c_{D_{\text{max}}+1}(b)$ using (4.9)
6. **for** $i = D_{\text{max}} + 1$ to $N$
7. Encode $x_i(b)$ according to the probability assignment $\hat{\theta}_{x_i(b)}$
8. Compute $c_{i+1}(b)$ using (4.10)
9. }
10. }

**Figure 4.9** Pseudocode for Phase II. The decoder is similar.

$2^{D_{\text{max}}} = O(N/B)$ leaf states, the aggregate time complexity of Line 6 of Figure 4.5 is $O(N \log(B)/B)$.

### 4.3.3 Pipelining

With pipelining, the adder tree can generate a result every $O(1)$ clock cycles. Pipelining is achieved by inserting registers between stages of the adder tree (Figure 4.10). Although pipelining the adder tree is simple in the data path, it complicates the control path of the *PMDL* routine because computing each $n_s^a$ value involves an $O(\log(B))$ delay.

The control aspects related to pipelining can be resolved by realizing that the CU needs to know $n_s^a$, not $\{n_s^b(b)\}_{b \in \{1, \ldots, B\}}$. Instead of receiving $\{n_s^a(b)\}_{a \in X, b \in \{1, \ldots, B\}}$ in Lines 3-5, and computing $n_s^0$ and $n_s^1$ in Line 6, we precompute all the symbol counts before invoking the *PMDL* routine. We first compute the block symbol counts on the $B$ PUs. We then loop through all depth-$D_{\text{max}}$ contexts and request each PU $b$ to send
\textbf{Figure 4.10} An adder tree in VLSI adds up $B$ terms in $O(\log(B))$ time. With pipelining, a symbol count is generated every $O(1)$ clock cycles.

$n_s^0(b)$ and $n_s^1(b)$ to the adder tree in parallel. We then assign the added up values to $n_s^0$ and $n_s^1$ when the adder tree has finished the computation. Furthermore, $n_s^0$ and $n_s^1$ have been precomputed, so Lines 3-5 can be removed. Because $2^{D_{\text{max}}+1} = O(N/B)$ symbol counts need to be computed, and the pipelined adder tree requires $O(1)$ time per outcome, precomputing symbol counts requires $O(N/B)$ time complexity. Pseudocode for the entire $O(N/B)$ complexity PTP-MDL encoder is outlined in Figure 4.11.

4.4 \textbf{Properties}

4.4.1 \textbf{Computational complexity and memory use}

\textbf{Computational complexity} - With our computational model, which assumes that computations performed with $2\log(N)$ bits of precision are $O(1)$, the time complexity of the PTP-MDL encoder includes the following components (i) computing the block
1. \textbf{in parallel} for $b \in \{1, \ldots, B\}$ 
   \begin{enumerate}
   \item Compute $\{n_s^a(b)\}_{a \in \mathcal{X}, \ s \in \mathcal{X}^{\nu_{max}}}$ /* Figure 4.4 */
   \end{enumerate}
2. 
3. 
4. Compute $\{n_s^a\}_{a \in \mathcal{X}, \ s \in \mathcal{X}^{\nu_{max}}}$ per (4.2) using a pipelined adder tree
5. Run $PMDL(\lambda)$ /* Figure 4.5; remove Lines 3-5 */
6. Describe $\hat{\mathcal{S}}$ and $\{r_s\}_{s \in \hat{\mathcal{S}}}$
7. \textbf{in parallel} for $b \in \{1, \ldots, B\}$ 
   \begin{enumerate}
   \item Send $\hat{\mathcal{S}}$ and $\{r_s\}_{s \in \hat{\mathcal{S}}}$ from the CU to PU $b$
   \item Run Phase II on PU $b$ /* Figure 4.9 */
   \end{enumerate}
8. 
9. 
10. 

\textbf{Figure 4.11} An outline of the $O(N/B)$ complexity PTP-MDL encoder.

symbol counts of all leaf contexts is $O(N/B)$ (see Section 4.2.4.1), (ii) adding up the block symbol counts with a pipelined adder tree is $O(N/B)$ (see Section 4.3.3), (iii) the $PMDL$ routine processes $O(2^{D_{max}}) = O(N/B)$ contexts and (after removing Lines 3-5) each function call is $O(1)$ (see Section 4.3.3), (iv) constructing the generator table $g$ is at most $O(N/B)$ (see Section 4.3.1), and (v) Phase II is $O(N/B)$ when context indices and $g$ are used to determine the generators of all the symbols. The decoder is also $O(N/B)$ because its structure is similar to that of Phase II (see Section 4.2.7). We conclude the discussion with the following theorem.

\textbf{Theorem 16} With computations performed with $2\log(N)$ bits of precision defined as $O(1)$ time, the Parallel Two-Pass MDL algorithm requires $O(N/B)$ time.

\textbf{Memory use} - With our computational model, which assumes that $\log(N)$ bit words are viewed as $O(1)$ memory, each PU $b$ of the PTP-MDL encoder stores (i) the block $x(b)$, which uses $O(N/B)$ memory, (ii) the block symbol counts, which use $O(2^{D_{max}+1}) = $
$O(N/B)$ memory, (iii) the generator table $g$, which uses $O(2^{D_{max}}) = O(N/B)$ memory, and (iv) $\tilde{S}$ and $\{r_s\}_{s \in \tilde{S}}$, which use at most $O(|\tilde{S}|) \leq O(2^{D_{max}+1}) = O(N/B)$ memory. The CU in the encoder also uses $O(N/B)$ memory. With $B$ PUs, the total memory use in the encoder is $O(N)$. The decoder also uses $O(N)$ memory because its structure is similar to that of Phase II. We conclude the discussion with the following theorem.

**Theorem 17** The Parallel Two-Pass MDL algorithm uses a total of $O(N)$ words of memory, which is $O(N \log(N))$ bits.

### 4.4.2 Redundancy analysis

The components of the redundancy w.r.t. $\hat{S}$ are as follows.

- The model redundancy of the natural code is at most $2|\hat{S}| - 1$ bits (see Section 4.2.3.4).
- The parameter redundancy introduced by two-part codes is $\frac{1}{2} \log(N) + O(1)$ bits per state (see Section 4.2.3.2). Note that the redundancy of two-part codes is within $1.221 + O(\frac{1}{N})$ bits per state of Rissanen’s bound.
- Each PU $b$ encodes the first $D_{max}$ symbols of $x(b)$ directly, which requires $D_{max} = \log(N/B) + O(1)$ bits (see Section 4.2.3.4).
- Because our computations are performed with finite precision, the outcome $\hat{S}$ of Phase I may not be an MDL tree source structure. Theorem 15 indicates that the coding length with $\hat{S}$ is at most $O(1)$ bits more than the coding length with the MDL structure $S^*$.
- The expected coding redundancy is $O(1)$ bits per PU (see Section 4.2.3.4).
- The implementation redundancy is less than $B' = o(B)$ bits (see Section 4.2.6).
Assuming that $S$ was the actual tree source structure that generated $x$, then $S \in D_{\text{max}}$ asymptotically. Therefore,

$$l_S^I + l_S^H \geq l_S^I + l_S^H.$$

Combining this observation with the redundancy components, we have the following theorem.

**Theorem 18** Neglecting coding redundancy, the pointwise redundancy of the Parallel Two-Pass MDL algorithm over the ML entropy of the input sequence $x$ w.r.t. the MDL tree source structure $S^*$ satisfies

$$\rho(x) \leq B \left[ \log \left( \frac{N}{B} \right) + O(1) \right] + \frac{|S^*|}{2} \left[ \log(N) + O(1) \right].$$

Including coding redundancy, if $x$ was generated by the tree source structure $S$, the expected redundancy of the Parallel Two-Pass MDL algorithm w.r.t. $S$ satisfies

$$\rho < B \left[ \log \left( \frac{N}{B} \right) + O(1) \right] + \frac{|S|}{2} \left[ \log(N) + O(1) \right].$$

When comparing the redundancy of PTP-MDL to naive parallelization, recall that the redundancy of a sequential algorithm for a length-$N/B$ block is approximately $\frac{|\hat{S}|}{2} \left[ \log(N/B) + O(1) \right]$ bits. Denoting the total redundancy of the naive approach and of PTP-MDL by $\rho_{\text{naive}}$ and $\rho_{\text{PTP}}$, respectively, we have

$$\lim_{N \to \infty} \frac{\rho_{\text{naive}}}{\rho_{\text{PTP}}} \approx \frac{B|\hat{S}|/2}{B + |\hat{S}|/2}.$$  

For $|\hat{S}| \gg B$, $\rho_{\text{naive}}/\rho_{\text{PTP}} \to B$. For $B \gg |\hat{S}|$, $\rho_{\text{naive}}/\rho_{\text{PTP}} \to |\hat{S}|/2 \ll B$. In practice, there is no actual tree source structure $S$ that generated $x$. Instead, $\hat{S}$ describes the statistics underlying $x$; $|\hat{S}|$ often increases almost linearly in $N$. In this case $\rho_{\text{naive}}/\rho_{\text{PTP}} \to B$. 

123
4.4.3 Tradeoffs between performance measures

The emphasis in this chapter has been a low-redundancy work-efficient algorithm. We also want to evaluate other system performance parameters, such as the delay, throughput, chip area, and power consumption (see Section 4.1.2). Because PTP-MDL only performs the actual coding in Phase II, and Phase I is $O(N/B)$, the delay is $O(N/B)$. The throughput is $O(B)$ because PTP-MDL processes a length-$N$ input sequence in $O(N/B)$ time, so scaling up $B$ increases the speed drastically. The chip area, which is proportional to the amount of logic and total memory used, is $O(N) + O(B)$ because the memory use is $O(N)$ (see Section 4.4.1) and we use $B$ PUs. Finally, the power consumption is also $O(N) + O(B)$ because the dynamic power consumption is $O(B)$, while the leakage power is proportional to the chip area, which is $O(N) + O(B)$. However, the constants in the $O(B)$ terms of the chip area and power consumption, which reflect entire computational units, are larger than the constants in the $O(N)$ terms, which reflect words of memory.

The number of blocks $B$ also affects the performance in some subtle ways.

- The redundancy of PTP-MDL is approximately $B \log(N/B) + \frac{|S|}{2} \log(N)$ bits, so increasing $B$ increases the redundancy. Another factor that degrades the compression quality further when $B$ is increased is that PTP-MDL approximates the MDL source w.r.t. the class $\mathcal{D}_{\text{max}}$ of tree source models. Increasing $B$ decreases $D_{\text{max}}$; hence $\mathcal{D}_{\text{max}}$ becomes smaller, and $\hat{S}$ may become less effective at describing $x$.

- PTP-MDL uses $O(N)$ memory regardless of $B$, but each PU stores some software, which uses a total of $O(B)$ memory. Therefore, although the data only uses $O(N)$ memory, the entire system’s memory use is $O(N) + O(B)$. Recall that the chip area and power consumption are also $O(N) + O(B)$. When $B$ is small and $N$ is large,
the $O(N)$ terms will dominate. For large $B$, the chip area and dynamic power scale linearly with $B$, which is unreasonable.

- **Work-efficient** algorithms [29] use $B$ computational units to solve problems of size $N$ in $O(N/B)$ time. As a consequence, for a work-efficient algorithm whose memory use is $O(N)$, the time×memory product is $O(N^2/B)$, meaning that increasing the degree of parallelization makes the solution more efficient w.r.t. the time×memory performance measure. A closer look at the complexity terms reveals that PTP-MDL requires $O(N/B) + O(1)$ time and uses $O(N) + O(B)$ memory, so the time×memory product is $O(N^2/B) + O(N) + O(B)$. For large $B$, the $O(B)$ term will dominate.

### 4.4.4 Random access compression

Willems et al. [66,67] have studied random access compression, wherein a decompression algorithm needs to be able to reconstruct portions of an input sequence without needing to decompress the entire compressed output. For example, a book may compress into a large file; with regular compression algorithms, it may be necessary for the entire file to be decompressed, even if the reader is only interested in a short passage.

The parallel source description that was developed in Section 4.2.3 supports a form of random access compression. If the user wants to retrieve information from block $x(b)$, there is no need to decompress the entire compressed output. Instead, the decompression algorithm only needs to read the source description of Phase I, whose length is $l_1^I$, and the compressed block generated by PU $b$, whose length is $l_1^I(b)$. We leave the analysis of the relations between PTP-MDL and random access compression for future research.
4.5 Extensions

4.5.1 Advanced parallel source description

In Section 4.2.3.3, we suggested using $K_s$ that does not depend on $n_s$. If $n_s < N$, we could save $\frac{1}{2} \log(N/n_s)$ bits by using a two-part code that is optimal for length-$n_s$ sequences. For an input $x$ generated by a fixed tree source $\{S, \Theta\}$, PTP-MDL will asymptotically identify $S$, and any extra coding length used for describing a coarse estimate of $n_s$ may be wasteful because $n_s$ will scale linearly with $N$. However, in practice $|\hat{S}|$ will increase with $N$, there will be many states with small $n_s$, and the potential for coding length savings will be large. Here we outline source description methods that describe coarse estimates of $n_s$ in order to decrease the coding length by using two-part codes with less bins.

We quantize $n_s$ to $\hat{n}_s$ where $\hat{n}_s$ belongs to a finite set $\{\xi_i\}$ of representation levels. In order to simplify the discussion, we assume that the $n_s \leq \hat{n}_s = \xi_i$, symbols generated by state $s$ are encoded using a two-part code that is optimal for length-$\xi_i$, sequences, hence the WCR is upper bounded by $\beta + \frac{1}{2} \log(\xi_i)$ bits per state $s$, where $\beta$ is some constant. We also assume that the same MDL source $\{\hat{S}, \hat{\Theta}\}$ is chosen, irrespective of the specific method used for describing $n_s$. Finally, we assume that $\xi_i = \sigma^i$ for some constant $\sigma$; hence the WCR with $\xi_i$ is upper bounded by $\beta + \frac{1}{2} \log(\sigma)$ bits. We validate this assumption by pointing out that increasing the number of representation levels by a factor of 2, which enables using $\sqrt{\sigma}$ instead of $\sigma$, may save $\frac{1}{2} \log(\sigma)$ bits per state in the two-part code, but requires 1 more bit for describing the representation level. Therefore, a geometrically increasing $\xi_i$ offers a useful tradeoff between the coding length of the two-part code and the description of the representation level.

**Probabilistic approach** - after $n_s$ is quantized to $\hat{n}_s = \xi_i$, we describe $\xi_i = \sigma^i$, with a coding length of $- \log(q_i \cdot \leq 1$. We model the likelihood that
$n_s$ is quantized to $\xi_i$ by $p_i$, then the expected redundancy satisfies

$$\rho \approx \sum_i p_i |\hat{S}| \left[ \beta + \frac{1}{2} \log(\xi_i) - \log(q_i) \right]$$

$$= |\hat{S}| \left[ \beta + \sum_i \frac{i \cdot p_i}{2} \log(\sigma) - \sum_i p_i \log(q_i) \right].$$

This expression is minimal when $q_i = p_i$; in this case

$$\rho \approx |\hat{S}| \left[ \beta + \frac{E[i]}{2} \log(\sigma) + H(\{p_i\}) \right],$$

where $H(\{p_i\}) \triangleq -\sum_i p_i \log(p_i)$ and $E[\cdot]$ denotes expectation w.r.t. $\{p_i\}$. Furthermore, $|\{\xi_i\}| = \log_\sigma(N) + O(1)$, so $H(\{p_i\}) \leq O(\log \log(N))$. Unfortunately, $\{p_i\}$ are unknown, so we cannot achieve the minimal expected redundancy. Instead, we offer several simple alternatives.

**Larger extra coding length for larger $n_s$** - we describe $\xi_i = \sigma^i$ with $i$ bits, where $i \geq 1$. This approach is intuitively appealing in practice because we expect that there are more states $s \in \hat{S}$ such that $n_s$ is small, and fewer states with large $n_s$, hence the implied $q_i$ is largest for small $i$. Because $n_s \leq \xi_i$, the WCR with $\xi_i$ is upper bounded by $\beta + i_s \left[ 1 + \frac{1}{2} \log(\sigma) \right]$ bits.

**Identical extra coding length for all $n_s$** - we describe $\xi_i$ with $\log(\log_\sigma(N))$ bits. For a state $s$ that generated $n_s$ symbols, where $n_s$ is quantized to $\tilde{n}_s = \xi_{i_s}$, the WCR with $\xi_{i_s}$ when using this approach is upper bounded by $\beta + \frac{i_s}{2} \log(\sigma) + \log(\log_\sigma(N))$ bits.

**Smaller extra coding length for larger $n_s$** - we describe $\xi_i = \sigma^i = N/\sigma^{\log_\sigma(N) - i}$ with $\lceil \log_\sigma(N) - i \rceil$ bits. The WCR with $\xi_i$ when using this approach is upper bounded by $\beta + i \left[ \frac{1}{2} \log(\sigma) - 1 \right] + \lceil \log_\sigma(N) \rceil$ bits. For $\sigma = 4$, all $n_s$ will have approximately the same per-state WCR, just like the approach of Section 4.2.3.3. Using $\sigma > 4$ gives states with larger $n_s$ larger per-state WCRs than states with smaller $n_s$. 

127
We leave the detailed study of these considerations, including how to incorporate the modified coding lengths during Phase I, for future research.

4.5.2 Practical aspects

The following issues are important in a practical parallel compression system.

- **Pipelining** - pipelining the adder tree complicates the control path of our algorithm. Instead, the time complexity for adding up the block symbol counts of $N/B$ states with an adder tree, but without pipelining, is $O(N \log(B)/B)$.

- **Memory-efficient decoder and Phase II** - in Section 4.3.1, we provided a work-efficient algorithm for determining the generators of all the symbols of $x$. A simpler approach stores $\hat{S}$ in a tree structure [29] and determines the current state by traversing the path from the root of the tree, following the last several symbols, until a leaf is reached. Storing $\hat{S}$ may require much less memory, but the typical complexity of this approach is $O(N/B \log(N/B))$.

- **Compression heuristics** - while the theoretical redundancy results predict competitive compression performance on large files, achieving similar results for shorter files would require studying the use of various heuristics for practical sources. We leave the evaluation of these practical aspects for future research.

4.5.3 Overlapping blocks

Suppose that $x$ is partitioned into overlapping blocks with overlap $D_{\text{max}}$, i.e., $x(1) = x_1^{N/B + D_{\text{max}}}$, $x(2) = x_2^{N/B + D_{\text{max}}}$, ..., $x(b) = x_b^{bN/B + D_{\text{max}}}$, ..., $x(B) = x_b^{N - N/B + 1}$. In this parallel compression framework, only the first block $x(1)$ has $D_{\text{max}}$ symbols with unknown context, and it seems that the excess redundancy above the bounds (1.1) and (1.2) is approximately $D_{\text{max}} + O(|\hat{S}|)$ bits instead of $B \cdot D_{\text{max}} + O(|\hat{S}|)$. In this compression
framework, the encoder can still compress $x$ in $O(N/B)$ time because PU $b$ can access the first $D_{\max}$ bits $x(b)$. Unfortunately, decoding block $b$ cannot begin to decompress $x(b)$ until the last $D_{\max}$ symbols of $x(b-1)$ have been decoded. Therefore, in a compression system with overlapping blocks, using parallel units in the decoder is not beneficial. We leave the analysis of PTP-MDL systems with overlapping blocks for future research.

### 4.6 Discussion

We compare the memory use, computational complexity, and redundancy performance of the Parallel Two-Pass MDL (PTP-MDL) algorithm to other source coding algorithms. Compressing a length-$N$ input sequence with serial source coding algorithms requires at least $O(N)$ time, but has potentially lower redundancies than PTP-MDL. The extra redundancy of PTP-MDL comes from (i) the $B \log \left( \frac{N}{B} \right)$ bits that describe symbols with unknown context at the beginning of each block, and (ii) PTP-MDL optimizes the MDL source w.r.t. $D_{\max}$, whereas serial source coding algorithms may consider unbounded context depths (see Chapter 2). Alternatively, we can compress each of $B$ blocks using a serial algorithm. This naive parallelization may attain $O(N/B)$ computational complexity, but its redundancy will be approximately $B$ times larger than the redundancy of the underlying serial algorithm on the entire length-$N$ input (see Section 4.4.2 for a comparison to the redundancy of PTP-MDL). With parallel CTW [66], the computational complexity is $O(N D_{\max}/B)$, where $D_{\max} = O(\log(N))$ is a reasonable choice [40]. Although the redundancy of parallel CTW was not analyzed, it seems safe to assume that it is comparable to the redundancy of PTP-MDL.

As with any novel method, PTP-MDL suggests a wide range of topics for future research. An important direction would examine the class of tree sources that is considered during the coding length minimization of Phase I. In particular, parallel compression al-
gorithms that support **unbounded context depths** may be important because deeper states may enable one to better approximate the underlying tree source model, leading to better compression in practice. In PTP-MDL, the simple structure of the full depth-$D_{\text{max}}$ tree allows us to pipeline the adder tree, process each internal node of the full tree in $O(1)$ time in Phase I, and construct the generator table $g$ efficiently in Phase II. Unfortunately, with unbounded context depths the computational complexity may increase because (i) the order in which states will be processed will be more complicated, hence pipelining may not work well, and (ii) we may not be able to construct $g$ efficiently. Therefore, future research on unbounded context depths must make sure that the increased modeling flexibility and compression improvement do not require slower processing.

**Single encoder** - consider a compression system that has several PUs that accumulate information on the input sequence in Phase I, but Phase II only has a single encoder. This may make sense if Phase I runs much slower than Phase II and we want to alleviate this bottleneck using parallelization. Furthermore, a single encoder may decrease the redundancy because only $D_{\text{max}}$ symbols with unknown context need be encoded directly. We leave the study of the single encoder case for future research.

Additional directions for future work were suggested before. First, mapping $B$ compressed blocks onto $B$ output lines is simple, but if the compressed blocks vary greatly in length, the implementation redundancy is much higher than the implementation redundancy of an interleaving scheme (see Section 4.2.6). An analysis of the implementation redundancy of this scheme would indicate whether interleaving is important. Second, a detailed analysis of the various tradeoffs involved in using PTP-MDL for random access compression, as suggested in Section 4.4.4, would be of interest. Third, the parallel source description methods of Section 4.5.1 are yet to be analyzed rigorously. Fourth, PTP-MDL compression systems with overlapping blocks have yet to be analyzed rigorously.
Finally, in this chapter we paid a price of roughly $B\log(N)$ bits for a factor-$B$ speedup. This can be viewed as an upper bound on the extra redundancy required for parallelization. However, it is not clear whether there is an inherent need for this extra redundancy. In order to conduct future research on parallel source coding in fruitful directions, it would be crucial to undertake a fundamental study of redundancy bounds for algorithms lying in different regions of the parallel design space.
CHAPTER 5

LOW COMPLEXITY SEQUENTIAL SUFFIX SORTING FOR BWT-BASED DATA COMPRESSION

5.1 Introduction

5.1.1 Suffix sorting and compression

Suffix sorting requires lexicographic ordering of all the suffixes of all the symbols in an input sequence. It was traditionally used for running queries on large texts. Recently it has received more research attention, owing to the appearance of the Burrows Wheeler transform (BWT) [21]. The BWT, which relies on suffix sorting, has attracted intense research interest because it achieves lossless compression results near the state of the art while being more efficient than other lossless compression methods [13, 42] in terms of computation and memory use.

Traditional BWT-based compression schemes first compute the BWT, whose output distribution is similar to piecewise i.i.d. (PIID) [22, 23, 25]. Then, they compress the BWT output with a lossless compression method designed for PIID sources [21–23, 25] (Figure 5.1). Because the divergence between the BWT output and a PIID distribution is quite small [25], compressing the BWT output with a PIID method yields good compression results [21, 22, 26].
Figure 5.1 A typical BWT-based compression system.

BWT-MDL is another direction in BWT-based compression. We use the BWT output only for estimating the tree source that leads to the minimum description length (MDL) with the input sequence. We then encode the input with this MDL tree source. The compression performance of BWT-MDL is $O(1)$ bits per tree source state above the lower bound on universal coding performance [2].

Despite their advantages, all BWT-based methods suffer from a common computational bottleneck: the suffix sorting needed to implement the BWT itself [22, 25]. Thus, the development of fast algorithms for suffix sorting has become crucial. In order to perform suffix sorting, it is necessary to store the entire database in memory. Therefore, suffix sorting algorithms must also use as little memory as possible. These two requirements, fast computation and efficient memory use, must both be satisfied in order for a suffix sorting method to be appealing.

5.1.2 Lossless compression in VLSI

An obvious way to speed up compression systems is by VLSI implementation. The majority of past work on fast lossless compression in hardware was concerned primarily with the VLSI aspects, thus outdated compression algorithms were used. For example, the implementation of Ziv and Lempel’s method [56] has attracted much attention [57, 58], as have implementations of the move to front algorithm [63]. Recent years have seen proposals for VLSI implementations of more advanced algorithms. For example,
Schindler [64] considers a bounded depth BWT variant, which simplifies his proposed VLSI implementation.

The obvious advantage of VLSI implementation is speed. However, many of the recent compression algorithms [13, 24, 42] are very complicated, and a VLSI implementation would probably be a major endeavor. Figure 5.1 shows a block diagram for a BWT-based system. Such a system might include a BWT block and a PIID compression block in the encoder, a PIID decompression block and an inverse BWT [21, 25, 26, 31] block in the decoder. The hardest part to implement in VLSI would probably be the BWT block, owing to the complicated structure of current algorithms. For example, the suffix trees method [3, 24, 32, 33, 36–38] for suffix sorting (see Section 1.3.4) grows a tree data structure with pointers to children nodes, short-cut pointers between nodes, etc. This is a complex data structure requiring sophisticated control. We conclude that, if we want to implement a high speed BWT-based compression system in VLSI, a suffix sorting method must feature more than fast computation and efficient memory use. The suffix sorting algorithm must be simple.

5.1.3 Overview of chapter

In this chapter, we propose a new suffix lists data structure that leads to several fast, sequential, and memory-efficient algorithms for suffix sorting. The remainder of the chapter is organized as follows. In the first part of the chapter, we present the fundamentals, and describe our basic algorithm. Section 5.2 lays down the suffix sorting problem formally, describes the state of the art in suffix sorting, and reviews the BWT from a suffix sorting perspective. Section 5.3 proposes to store suffixes that have been processed so far in a linked list in sorted order. Several observations on properties of these suffix lists, and their relation to the BWT output, are made. In particular, the suffix list needs to be modified in a single position that can be determined by scanning the BWT
output until we match a symbol for which we are searching. These observations lead to
the basic algorithm, whose details are presented along with an analysis of its memory use
and computational complexity. While its typical complexity for length-$N$ inputs over a
size-$|\mathcal{X}|$ alphabet $\mathcal{X}$ is conjectured to be $O(|\mathcal{X}|N)$, its worst-case behavior is proved to
be $O(N^2)$.

In the second part of the chapter, we present fast extensions to the basic algorithm.
Section 5.4 proposes to improve the basic algorithm by scanning the BWT output for a
match in both directions. We analyze the computational complexity of this bidirectional
algorithm, and prove that its worst-case complexity is $O\left(|\mathcal{X}|N \log \left( \frac{N}{|\mathcal{X}|} \right) \right)$. We further
extend this bidirectional approach in Section 5.5, where we suggest using two hierarchies
of lists. If the match in the BWT output is nearby, we scan for it in the lower suffix list,
else we use a sparse upper list for accelerated access. The worst-case complexity of this
hierarchical algorithm is proved to be $O\left(N \sqrt{|\mathcal{X}| \log \left( \frac{N}{|\mathcal{X}|} \right)} \right)$.

The third and final part of the chapter is oriented towards practical aspects. Section
5.6 presents further enhancements and applications for suffix lists. First, we show
that our algorithms can be accelerated by using a reduced input alphabet. Then, we fur-
ther exploit the truly distinguishing feature of suffix lists - their simplicity. We describe a
proposed VLSI implementation of the bidirectional algorithm; the potential application
to a high speed BWT-based compression system in VLSI is also discussed. Simulation re-
results for the basic, bidirectional, and hierarchical algorithms are described in Section 5.7.
The results indicate that software implementations of suffix list algorithms are competi-
tive in terms of speed with other suffix sorting methods, and are especially fast for inputs
over small alphabets, such as DNA sequences. Finally, Section 5.8 presents conclusions
and offers topics for future work.
5.2 Suffix Sorting and the BWT

5.2.1 Suffix sorting

Following Section 1.3.2, we append a unique sentinel symbol $\$ at the end of the sequence, i.e., $x_{N+1} = \$$, and assume that $\$$ is larger lexicographically than any symbol in $\mathcal{X}$. We define a suffix as $s_i \triangleq x_i^{N+1}$ for $1 \leq i \leq N + 1$. We determine the lexicographic order of suffixes by comparing symbol to symbol, from low to high indices, and we use the terms larger and smaller for suffixes and symbols in a lexicographic sense. By the properties of the sentinel symbol, all suffixes are uniquely ordered. The problem of suffix sorting can now be defined formally as determining the single sorted order (in increasing order) of all suffixes $s_i$, $1 \leq i \leq N + 1$.

**Example 7** Suppose $\mathcal{X} = \{0, 1\}$, $N = 4$, and $x = 1100$. We append the sentinel to $x$ and get $1100\$$. The suffixes are $s_1 = 1100\$$, $s_2 = 100\$$, $s_3 = 00\$$, $s_4 = 0\$$, and $s_5 = \$$. Clearly, $00\$ < $0\$ < $100\$ < $1100\$ < $\$$, and the sorted order of the suffixes is $(s_3, s_4, s_2, s_1, s_5)$.

5.2.2 State of the art

One of the earliest results on suffix sorting was by McCreight [37], who showed how a suffix tree, a tree structure pointing to suffixes in the input sequence, can be constructed in time that is linear in the length of the sequence (see Section 1.3.4). However, the data structure was quite cumbersome, and the memory use was large. Recent years have seen improvements in suffix tree algorithms. Larsson [33] showed a suffix tree algorithm that enables deleting symbols from the tree efficiently, thus supporting sliding window compression algorithms such as Ziv-Lempel [56]. Kurtz [32] showed how to implement suffix trees with low memory use, typically around 10 bytes per input symbol. Finally, Ukkonen [38] described a sequential suffix tree construction. For a comprehensive survey
of suffix tree algorithms, see Giegerich and Kurtz [36]. All the suffix tree algorithms have linear worst-case complexity. In practice, however, they are slower than other suffix sorting methods on typical inputs because they maintain complicated data structures.

Another direction in suffix sorting was provided by Bentley and Sedgewick [68]. Instead of using a specialized method for suffix sorting, they perform a ternary-split, where suffixes headed by identical symbols are compared based on their following symbols. An advantage of this method is that it uses only 5 bytes per input symbol. Furthermore, it is very fast when the average matching length (AML), the average number of symbols evaluated until a difference between adjacent suffixes is identified, is low. However, the worst-case complexity of the ternary-split method is $O(N^2)$. When the AML is high, the suffix tree algorithms are better, owing to their linear worst-case complexity.

In the doubling method, introduced by Karp et al. [69], after pass $i$ the symbols are sorted according to their depth-$2^i$ suffixes. Thus, the number of passes on the sequence cannot be too large. Larsson and Sadakane [34] refined the doubling method by not running the sort on suffixes that have already been sorted, and by improving the data structures. Their improved algorithm has $O(N \log(N))$ worst-case computational complexity, and simulation results indicate robust performance in any AML range. Lastly, their algorithm uses 9 bytes per input symbol.

The ternary-split algorithms can also be incorporated as a low-level routine in more powerful algorithms, as proposed by Seward [35]. By avoiding unnecessary comparisons and improving the memory access behavior of the algorithm, Seward achieves substantial speedups. In practice, his sorting algorithm is faster than the other methods [32, 34, 68].

Finally, we note that, among these algorithms, only some algorithms for constructing suffix trees perform suffix sorting sequentially [36, 38]. In contrast, algorithms based on the doubling or ternary-split methods process the entire input sequence in several
iterations. Our suffix lists method, which is presented in the remainder of this chapter, is sequential.

5.2.3 A suffix sorting perspective on the BWT

Following Section 1.3, the BWT output $y$ is obtained by first running prefix sorting, and then scanning the prefixes in sorted order, and outputting for each prefix the following symbol in the input sequence. Prefix sorting is equivalent to suffix sorting, except that we begin by reversing the order of the input sequence. Therefore, $y$ can be generated by first reversing the order of the sequence, then using suffix sorting, scanning through the sorted suffixes, and outputting $x_i$ for every $s_{i+1}$. However, $s_1$ does not have a preceding symbol; we output the BWT index that indicates its position in $y$ instead.

Example 8 Suppose $\mathcal{X} = \{0, 1\}$, $N = 4$, and $\hat{x} = 0011$. Reversing the order yields $x = 1100$, as in Example 7. The sorted order of the suffixes is $(s_3, s_4, s_2, s_1, s_5)$, and the preceding symbols are $(x_2, x_3, x_1, x_?, x_4)$, where $x_?$ is unknown. The BWT output, without $x_?$, is 1010, and the BWT index is 4.

Additional properties of the BWT, including the existence of the inverse BWT and the BWT output distribution, were described in Section 1.3.

5.3 Basic Algorithm

After introducing suffix lists in Section 5.3.1, we provide several observations that lead to the key idea of the chapter and present the core of the algorithm in Section 5.3.2. Section 5.3.3 examines three cases arising in the core of the algorithm, Section 5.3.4 outlines the memory use, and Section 5.3.5 discusses the computational complexity.
5.3.1 Suffix lists

In the remainder of this chapter, we consider a method for suffix sorting that sequentially inserts the suffixes from $s_{N+1}$ down to $s_1$, i.e., in reverse order, into a sorted list of previously processed suffixes. We call this data structure a suffix list. We call the largest suffix in the suffix list the head, and the smallest the tail. Scanning increasing elements of the list towards the head will be called scanning up the suffix list. Scanning decreasing elements towards the tail will be called scanning down the list.

In a doubly linked suffix list (Figure 5.2), each element holds, in addition to the data, previous and next pointers to the previous and next elements in the suffix list. We also keep pointers to the head and tail of the suffix list.\footnote{Because the sentinel symbol is larger than all symbols of $\mathcal{X}$, the sentinel suffix $s_{N+1}$ is always the largest suffix. Therefore, in some implementations the head need not be stored.} This structure allows for moving in either direction in the list, and for inserting an element before or after a given element using only local updates. However, each of the next and previous pointers uses 4 bytes on current typical computer architectures; an implementation that is more memory-efficient is discussed in Section 5.5. Note that element $i$ in the suffix list does not contain the suffix $s_i$ itself. Rather, $s_i$ can be determined by considering the symbols from $x_i$ up to $x_{N+1}$.

5.3.2 The core of the algorithm

In Section 5.2 we mentioned that, after adding the unique sentinel symbol at the end of the sequence, all suffixes are uniquely ordered. Once $k$ suffixes have been sorted, adding another suffix will not change their relative order. This leads to the key idea in this chapter.
Figure 5.2 Structure of a doubly linked suffix list. The basic algorithm scans upwards, examining increasing suffixes, up to the head.

Observation 1 By keeping suffixes that have been processed so far sorted in a suffix list, after having processed $k$ suffixes, processing the next suffix requires modifying the list only where the new suffix is added.

We process the input sequence sequentially in reverse order, i.e., from $x_N$ down to $x_1$. Given a new input symbol $x_i$, $s_i = x_is_{i+1}$. Consider $s_j$ such that $j > i$, so $s_j$ is already in the suffix list. We can have $s_j > s_i$ either if $x_j > x_i$, or if $x_j = x_i$ and $s_{j+1} > s_{i+1}$. Define

$$S_i \triangleq \{ s_j : j > i, x_j = x_i, s_j > s_i \}$$

$$= \{ s_j : j > i, x_j = x_i, s_{j+1} > s_{i+1} \},$$

the set of suffixes already in the list that are larger than $s_i$ and match $s_i$ in the first symbol.
Observation 2 If $S_i$ is nonempty, $s_i$ should be added to the linked list just before the smallest element of $S_i$.

If $S_i$ is nonempty, its smallest element is the smallest suffix $s_j$ that satisfies $s_{j+1} > s_{i+1}$ and $x_j = x_i$. It can be found easily because the position of $s_{i+1}$ in the suffix list is known (it is the most recently added suffix), and we can scan up the list (examining increasing suffixes) until such a $s_{j+1}$ satisfying $x_j = x_i$ is found. We call this a match for $x_i$. The upward scan through the suffix list while scanning for a match is the core of the algorithm.

While processing $x_i$, the intermediate BWT output, a permutation of $x_{i}^{N}$ denoted $y^{i}$, is defined over the suffixes that have been sorted up to that point. Given a suffix list $(s_{k_1}, s_{k_2}, \ldots, s_{i-1}, \ldots, s_{j+1}, \ldots, s_{k_{N-i+1}})$, the corresponding intermediate BWT output is $y^{i} = (x_{k_1-1}, x_{k_2-1}, \ldots, x_{i}, \ldots, x_{j}, \ldots, x_{k_{N-i+1}-1})$. The terms up, down, head, and tail concerning the intermediate BWT output are defined correspondingly.

Observation 3 Because scanning for a match at step $i$ consists of scanning the symbols $x_j$ preceding the sorted suffixes $s_{j+1}$ for $s_{j+1} > s_{i+1}$, it is equivalent to scanning up the intermediate BWT output $y^{i}$.

When $S_i$ is nonempty, we will find a match $x_j$ for $x_i$. After finding a match, we need to add $s_i$ to the list just before $s_j$. We also check whether $s_j$ was the tail; if it was, we update the tail accordingly. Alternatively, when $S_i$ is empty we will scan all the way up the suffix list until we reach the head. We now explain the circumstances when this happens.

5.3.3 Three cases

- Case 1: when we find a match $x_j$ for $x_i$, we update the suffix list and the tail accordingly.
• **Case 2**: if \( x_i \) has not appeared in \( x_{i+1}^N \), i.e., it is a new symbol, \( S_i \) is empty and it is impossible to find a match no matter how far up the list we scan.

• **Case 3**: if the largest suffix \( s_{j+1} \) in the list that satisfies \( x_i = x_j \) is smaller than \( s_{i+1} \), \( S_i \) is again empty, and when scanning up the list we will not find a match.

Among the three cases, Case 1 is the most common. When Cases 2 or 3 occur, \( S_i \) is empty, and we could expend considerable computational resources scanning all the way up to the head.

Case 2 can be managed by maintaining *symbol arrays*. For each symbol in \( \mathcal{X} \), we keep track of whether it has already appeared. When \( x_i \) has not already appeared, it is a new symbol, and we search for the largest symbol \( \alpha \) that has appeared and that also satisfies \( \alpha < x_i \). If such an \( \alpha \) exists, \( s_i \) is added right after the largest suffix \( s_j \) headed by \( \alpha = x_j \) (the largest symbol that is smaller than \( x_i \) and has appeared). This is Case 2A. If no such \( \alpha \) exists, \( x_i \) is smaller than all the symbols that have already appeared; thus, \( s_i \) is smaller than all the suffixes already in the suffix list, and we put \( s_i \) just before the tail, and update the tail accordingly. This is Case 2B. We call the resolution of Case 2 an *adding a new symbol*.

We support adding a new symbol by maintaining arrays that for each symbol in \( \mathcal{X} \) have entries that determine whether that symbol has appeared and, if it has, the largest suffix that it heads. We call these symbol arrays *appeared* and *largest*; their memory use is \( O(|\mathcal{X}|) \). The computation involved in checking for every symbol in \( x \) whether it has appeared is \( O(N) \). The computation for adding new symbols is \( O(|\mathcal{X}|^2) \) because we add new symbols at most \( |\mathcal{X}| \) times, and each time we process less than \( |\mathcal{X}| \) entries in the symbol arrays (usually \( O(N) \) dominates \( O(|\mathcal{X}|^2) \)). These requirements are smaller than the memory use for the suffix list and the computational resources expended on Cases 1 and 3.
For practical purposes, when $N$ is large, new symbols appear early on in the sequence. Instead of expending a total of $O(N)$ operations on checking for every symbol in the input sequence whether it has appeared, we might be better off letting the algorithm scan all the way up the suffix list on Case 2. With $|X|$ symbols, the worst-case aggregate number of operations for this procedure is $O(|X|N)$. Alternatively, because $|X|$ is known, once all the symbols have appeared we can stop checking. However, many sources, text files in particular, use only part of the alphabet, so this latter approach is often impractical. We conclude that Case 2 can be solved in linear complexity, and concentrate on Cases 1 and 3 in the remainder of the chapter.

In Case 3, $S_i$ is empty although $x_i$ has already appeared. After reaching the head, we need to add $s_i$ right after $\text{largets}_{x_i}$. Having added $s_i$ to the suffix list, it is the largest suffix headed by $x_i$ in the list, and we need to update the $\text{largets}$ array. Although the algorithmic solution for Case 3 is different than that for Case 1, reaching the head indicates where to add the suffix, and is similar to a match. Therefore, we analyze the performance of Cases 1 and 3 together in the remainder of the chapter. This discussion leads to the basic algorithm; pseudocode for it appears in Figure 5.3.

Example 9  Suppose $X = \{0, 1\}$, $N = 4$, and $x = 1100$. We append the sentinel to $x$ and get 1100$. Following each iteration $i$ we will show the intermediate BWT output $y^{i-1}$ stacked above the suffix list (we stack $x_i$ above $s_{i+1}$), and indicate the beginning and end of a scan for a match by an arrow and asterisk, respectively.

We initialize the list with $\binom{0}{s_0}$ (the sentinel suffix) and initialize $\text{appeared}_0 \leftarrow \text{TRUE}$, $\text{appeared}_0 \leftarrow \text{FALSE}$, $\text{appeared}_1 \leftarrow \text{FALSE}$, head $\leftarrow 5$, tail $\leftarrow 5$, and $\text{largest}_8 \leftarrow 5$.

For $i = 4$, $c \leftarrow x_i = 0$, $\text{appeared}_0 = \text{FALSE}$, and we have a new symbol. But $c = 0$ is less than $\$, the only symbol that has appeared. This is Case 2B. We put $s_i = s_4$ before
Initialize symbol arrays, and initialize suffix list with $s_{N+1}$

for $i = N$ down to $1$

\[
c \leftarrow x_i
\]

if ($\text{appeared}_c = \text{FALSE}$) { /* Case 2 */
    Search for the largest $\alpha$ such that $\alpha < c$ and $\text{appeared}_\alpha = \text{TRUE}$
    if ($\alpha$ exists) { /* Case 2A */
        Add $s_i$ to the list after $\text{largest}_\alpha$
    } else { /* Case 2B */
        Add $s_i$ to the list before $\text{tail}$, update $\text{tail} \leftarrow i$
    }
    $\text{appeared}_c \leftarrow \text{TRUE}$, $\text{largest}_c \leftarrow i$
} else { /* $c$ has appeared */
    $\text{temp} \leftarrow \text{next}_{i+1}$
    while ($x_{\text{temp}-1} \neq c$ and $\text{temp} \neq \text{head}$) {
        $\text{temp} \leftarrow \text{next}_{\text{temp}}$
    } if ($x_{\text{temp}-1} = c$) { /* Case 1 */
        Add $s_i$ to the list before $s_{\text{temp}-1}$
        if ($\text{tail} = \text{temp} - 1$) { Update $\text{tail} \leftarrow i$
    } else { /* Case 3 */
        Add $s_i$ to the list after $\text{largest}_c$, update $\text{largest}_c \leftarrow i$
    } /* Cases 1 and 3 */
} /* $c$ has appeared */
} /* for loop */

**Figure 5.3** Pseudocode for the basic algorithm. Global variables are $\text{head}$, $\text{tail}$, and $\text{temp}$ pointers, the $i$ index, and $c$ and $\alpha$, which are symbols.

$s_{\text{tail}} = s_5$, initialize $\text{largest}_0 \leftarrow 4$, and update $\text{appeared}_0 \leftarrow \text{TRUE}$ and $\text{tail} \leftarrow 4$. The suffix list becomes $\left( \overline{3} 0 \overline{5} \right)$.  

For $i = 3$, $c \leftarrow 0$, and $\text{appeared}_0 = \text{TRUE}$. We enter the list at $s_{i+1} = s_4$ and try to match $c = 0$ (scanning in the direction of the arrow). The first suffix $s_{j+1}$ that satisfies $x_j = 0$ is $s_5$ (indicated by an asterisk). This is Case 1. We put $s_i = s_3$ before $s_j = s_4$, since $\text{tail} = 4 = j$ we update $\text{tail} \leftarrow 3$, and the list becomes $\left( \frac{1}{1} 0 \overline{3} 0 \overline{5} \right)$. 

For $i = 2$, $c \leftarrow 1$, $\text{appeared}_1 = \text{FALSE}$, and we have a new symbol. The symbol $\alpha = 0$ satisfies $\alpha < c$ and $\text{appeared}_\alpha = \text{TRUE}$. This is Case 2A. Since $\text{largest}_\alpha = \text{largest}_0 = 4$, we put $s_i = s_2$ after $s_{\text{largest}_0} = s_4$, initialize $\text{largest}_1 \leftarrow 2$, update $\text{appeared}_1 \leftarrow \text{TRUE}$, and the list becomes $\left( \frac{1}{1} 0 \overline{3} 0 \overline{5} \right)$.
For $i = 1$, $c \leftarrow 1$, and $appeared_1 = TRUE$. We enter the suffix list at $s_{i+1} = s_2$ and try to match $c = 1$. We do not find a match. This is Case 3. Since $largest_1 = 2$, we put $s_i = s_1$ after $s_2$, update $largest_1 \leftarrow 1$, the list becomes $\left( \begin{array}{c} s_3 \ s_4 \ s_2 \ s_1 \ s_0 \end{array} \right)$, and the BWT index is 4. These results are identical to those from previous examples.

### 5.3.4 Memory use

The data structures used in the pseudocode of Figure 5.3 include a linked list, symbol arrays, and global variables. For each suffix $s_i$, its element in the linked list consists of $next_i$ and $previous_i$ pointers, and the $x_i$ symbol. The elements of the list are arranged in an array, so $x_i$, $x_j$, etc., can be accessed by first accessing element $i$ or $j$, respectively, in the array, and then accessing the respective symbols in these elements.\(^2\) For current typical software implementations these use $9N$ bytes (4 bytes for each pointer and 1 byte for the symbol); the structure of the suffix list appears in Figure 5.2. For each symbol in the alphabet, the symbol arrays contain $appeared$ and $largest$ entries. For current typical implementations these use $5|\mathcal{X}|$ bytes (4 for each pointer and 1 for the Boolean variable). The global variables consist of the $head$, $tail$, and $temp$ pointers, the $i$ counter, and $c$ and $\alpha$, which are symbols. An implementation that is more memory-efficient is discussed in Section 5.5.

### 5.3.5 Computational complexity

The **while** statement in the pseudocode of Figure 5.3 is the only portion that could require more than $O(N)$ computational complexity. The number of **while** loops is proportional to the number of accesses to the list for Cases 1 and 3. Therefore, we define

\(^2\)In Section 5.3.5 we will explain that the **while** statement in the pseudocode of Figure 5.3 is the most computationally sensitive portion of the algorithm. Because we access $x_{temp-1}$, the implementation can be optimized by storing $x_{i-1}$ in each element instead of $x_i$. 

145
the computational complexity of our algorithm according to the aggregate number of accesses to the suffix list.

The remainder of the section analyzes the aggregate number of accesses to the list. In Section 5.3.5.1, we explain\(^3\) that for typical input sequences generated by tree sources, the computational complexity is \(O(|X|N)\). Then, Section 5.3.5.2 proves that the worst-case complexity is \(O(N^2)\).

5.3.5.1 Typical-case

In Section 1.3 we discussed tree sources, and explained that symbols in the BWT output are sorted according to states in the tree, and each of those segments is distributed similar to i.i.d. Because the intermediate BWT output \(y^i\) is identical to the BWT on the sequence \(x^N_{i+1}\), the distribution of \(y^i\) is likewise similar to piecewise i.i.d. (PIID). The following result, from Effros et al. [25], bounds the divergence between these distributions, and leads to Approximation 1.

**Proposition 1 ([25])** The divergence between the BWT output and a PIID distribution is at most \(O\left(\sqrt{N}\log(N)\right)\).

Although this distribution is not PIID in a strict sense [25], in this subsection we approximate it as such in order to present our intuition concerning the typical computational complexity of the basic algorithm.

**Approximation 1** For tree sources, the intermediate BWT output \(y^i\) has a PIID distribution.

We now analyze the aggregate number of accesses to the list required by the basic algorithm, using Approximation 1. Owing to Observation 3, scanning for a match is\(^3\)This is not a proof, and should be viewed as an intuitive explanation. The \(O(|X|N)\) results, which are conjectures, will be verified by simulation in Section 5.7.
equivalent to scanning \( y^i \). Let segment, i.e., the group of symbols in \( y^i \) generated by state \( s \) of the tree source, be the initial segment in which the upwards scan for a match to \( x_i = \alpha \) begins, and let \( \nu_s^\alpha(i) \) denote the number of accesses to the list in this case. We define \( \nu_s^\alpha(i) \triangleq 0 \) if \( x_i \neq \alpha \) or \( x_i \not\in \text{segment}_s \). We call the event in which a scan for a match fails in \( \text{segment}_s \), and moves into the next segment, a spill, and denote the event that \( x_i = \alpha \) spilled out of the initial \( \text{segment}_s \) by \( \text{spill}_s^\alpha(i) \). A spill is treated algorithmically either as Case 1 when there is a match later on, or as Case 3 when there is no match; we consider how spills affect the performance later. We can now write

\[
E[\nu_s^\alpha(i)] = \frac{\mu_s^\alpha(i)}{E[\nu_s^\alpha(i) \mid \text{match in segment}_s] \cdot p(\text{match in segment}_s)} + \frac{\nu_s^\alpha(i)}{E[\nu_s^\alpha(i) \mid \text{spill}_s^\alpha(i)] \cdot p(\text{spill}_s^\alpha(i))}.
\]

(5.1)

If the match is in \( \text{segment}_s \), \( \nu_s^\alpha(i) \) is geometrically distributed, because \( \text{segment}_s \) is approximated as i.i.d. Therefore,

\[
\mu_s^\alpha(i) \leq \frac{1}{p(\alpha \mid s)}.
\]

Denoting the expected length of \( \text{segment}_s \) as \( q_sN \), where \( \sum_{s \in \mathcal{S}} q_s = 1 \), the expected number of appearances of \( \alpha \) in \( \text{segment}_s \) is \( q_sp(\alpha \mid s)N \); hence,

\[
\sum_{i=1}^{N} \mu_s^\alpha(i) \leq [q_sp(\alpha \mid s)N] \left[ \frac{1}{p(\alpha \mid s)} \right] = q_sN.
\]

Summing over all the segments,

\[
\sum_{s \in \mathcal{S}} \sum_{i=1}^{N} \mu_s^\alpha(i) \leq \sum_{s \in \mathcal{S}, p(\alpha \mid s) > 0} q_sN \leq N,
\]

where the second inequality holds (with equality) when \( p(\alpha \mid s) > 0 \) for each \( \text{segment}_s \), \( s \in \mathcal{S} \). The expected aggregate number of accesses for symbols matched within their own segment over all the symbols is therefore at most

\[
\sum_{\alpha \in \mathcal{X}} \sum_{s \in \mathcal{S}} \sum_{i=1}^{N} \mu_s^\alpha(i) = \sum_{\alpha \in \mathcal{X}} \sum_{s \in \mathcal{S}, p(\alpha \mid s) > 0} q_sN \leq |\mathcal{X}|N.
\]

(5.2)
Consider now the case where the scan for a match spills into the next segment. This happens only when \( x_i = \alpha \) is the symbol \( \alpha \) closest to the head in \( \text{segment}_s \), and thus the last \( \alpha \) to be encountered in the upwards scan. Accordingly, in this case we say that \( x_i = \alpha \) is the last \( \alpha \) in \( \text{segment}_s \). The position of \( x_i \) in \( \text{segment}_s \) depends on the first several symbols of \( s_i \). These symbols were generated by the same tree source (assumed to be ergodic and stationary) that generated the suffixes already in the list. Therefore, the following approximation is reasonable.

**Approximation 2** At stage \( i \), the position of \( x_i \) is uniformly distributed over the intermediate BWT output \( y^i \).

If \( x_i = \alpha \) is the last \( \alpha \) in \( \text{segment}_s \), the upwards scan will not match \( x_i \) in \( \text{segment}_s \), and there is a spill. Using Approximation 2, the position of \( x_i = \alpha \) in \( \text{segment}_s \) is uniformly distributed. It follows that if there are \( k \) appearances of \( \alpha \) in \( \text{segment}_s \), the probability that \( x_i = \alpha \) is last, and there is a spill, is \( \frac{1}{k} \). At the end of the algorithm the expected length of \( \text{segment}_s \) is \( q_s N \), so the expected number of appearances of \( \alpha \) in \( \text{segment}_s \) is \( q_s p(\alpha|s) N \), and the expected number of spills for symbol \( \alpha \) in \( \text{segment}_s \) is approximately

\[
\sum_{i=1}^{q_s p(\alpha|s) N} \frac{1}{i} = \ln(N) + O(1). \tag{5.3}
\]

Consider a scan for a symbol \( x_i = \alpha \) that spills into \( \text{segment}_s \) with \( p(\alpha|\bar{s}) = 0 \), so that \( \alpha \notin \text{segment}_s \). In this case we need to scan through all of \( \text{segment}_s \), which requires a large number of accesses. To quantify this, we denote the set of segments that contain \( \alpha \in \mathcal{X} \) by \( \mathcal{S}^\alpha \), and the individual segments in \( \mathcal{S}^\alpha \) by \( \text{segment}_{\mathcal{S}^\alpha(1)}, \text{segment}_{\mathcal{S}^\alpha(2)}, \ldots, \text{segment}_{\mathcal{S}^\alpha(|\mathcal{S}^\alpha|)} \).

Using our previous terminology, we denote the probability that a symbol belongs to \( \text{segment}_{\mathcal{S}^\alpha(j)} \) by \( q_{\mathcal{S}^\alpha(j)} \), where \( j \in \{1, 2, \ldots, |\mathcal{S}^\alpha|\} \). Following each segment in \( \mathcal{S}^\alpha \), there is a (possibly empty) set of contiguous segments that do not contain \( \alpha \); we denote the contiguous set following \( \text{segment}_{\mathcal{S}^\alpha(j)} \) by \( \text{segment}_{\mathcal{S}^\alpha(j)} \), and its probability by \( \bar{q}_{\mathcal{S}^\alpha(j)} \). Because
the segments are ordered, we have

$$\sum_{j=1}^{|S^\alpha|} q_{S^\alpha(j)} + \overline{q}_{S^\alpha(j)} \leq 1.$$  

We will have $s \in S^\alpha$ only if $q_s > 0$ and $p(\alpha|s) > 0$. Therefore,

$$\sum_{j=1}^{|S^\alpha|} \overline{q}_{S^\alpha(j)} < 1. \quad (5.4)$$

Now, the probability for being in $S_{S^\alpha(j)}$ and generating $\alpha$ is

$$p(\text{segment}_{S^\alpha(j)}, \alpha) = q_{S^\alpha(j)} p(\alpha|S^\alpha(j)). \quad (5.5)$$

When processing $x_i$, having already processed $N - i$ symbols, if $x_i = \alpha$ and $x_i \in \text{segment}_{S^\alpha(j)}$, the probability for a spill is

$$p(\text{spill}_{S^\alpha(j)}(i)|x_i = \alpha, x_i \in \text{segment}_{S^\alpha(j)}) \approx \frac{1}{q_{S^\alpha(j)} p(\alpha|S^\alpha(j))(N - i)}, \quad (5.6)$$

where the approximation is due to substituting the expected length of the segment for the real length. The expected number of required accesses satisfies

$$E[\nu_{S^\alpha(j)}(i)|\text{spill}_{S^\alpha(j)}(i)] \approx (N - i)\overline{q}_{S^\alpha(j)} + \frac{1}{q_{S^\alpha(j+1)}}, \quad (5.7)$$

where the second term is caused by scanning $\text{segment}_{S^\alpha(j+1)}$. The expected number of spills for symbol $\alpha$ in $\text{segment}_{S^\alpha(j)}$ was shown in (5.3) to be approximately $\ln(N)$. Summing over all the spills, the expected aggregate number of accesses indicated by the second term is $O(\log(N))$. In contrast, we will now show that the first term has a larger impact on the computation. Putting (5.1), (5.5), (5.6), and (5.7) together, the expected aggregate number of accesses caused by spills from $\text{segment}_{S^\alpha(j)}$ while searching for $\alpha$ satisfies

$$\sum_{i=1}^N \nu_{S^\alpha(j)}(i) \approx \sum_{i=1}^N \frac{[q_{S^\alpha(j)} p(\alpha|S^\alpha(j))] [(N - i)\overline{q}_{S^\alpha(j)}]}{q_{S^\alpha(j)} p(\alpha|S^\alpha(j))(N - i)}$$

$$= \sum_{i=1}^N \overline{q}_{S^\alpha(j)} = N \overline{q}_{S^\alpha(j)}. \quad (5.8)$$

149
When \( \overline{q} S^{*}(j) > 0 \) for some \( j \in \{1, 2, \ldots, |S^{*}|\} \), (5.8) will be \( O(N) \). Summing over all segments that can contain \( \alpha \), and using (5.4),

\[
\sum_{s \in S} \sum_{i=1}^{N} \xi_{s}^{a}(i) \approx \sum_{j=1}^{|S^{*}|} N \overline{q} S^{*}(j) < N.
\]

Summing over all the symbols, the expected aggregate number of accesses caused by spills is

\[
\sum_{a \in \mathcal{X}} \sum_{s \in S} \sum_{i=1}^{N} \xi_{s}^{a}(i) \leq \sum_{a \in \mathcal{X}} \sum_{a} N = |\mathcal{X}|N.
\]

(5.9)

It follows that the expected aggregate number of accesses to the list caused by spills is less than \( |\mathcal{X}|N \) to first order. Using (5.2), the expected aggregate number of accesses for symbols matched within their own segment is at most \( |\mathcal{X}|N \). Therefore, the expected aggregate number of accesses to the list is less than \( 2|\mathcal{X}|N \).

Tree sources can be represented as Markov sources, and if the source is irreducible and aperiodic, large deviation results can be applied [70]. In this case the aggregate number of accesses to the list will be similar to the expected values asymptotically, except for a set of input sequences with vanishing probabilities. We summarize this discussion with a conjecture.

**Conjecture 3** For irreducible aperiodic tree sources, the typical computational complexity of the basic algorithm is at most \( O(|\mathcal{X}|N) \).

It is well known that general-purpose sorting algorithms have \( O(N \log(N)) \) computational complexity [29]. The method described here is more efficient because, when we insert a new suffix into the list, it consists of a symbol followed by a suffix, which is already in the list. The only new information in the suffix is the additional symbol, and the rest of the suffix has already been processed. Nonetheless, as discussed in Section 5.3.5.2, this does not guarantee low computational complexity for every input sequence.
5.3.5.2 Worst-case

**Theorem 19** The worst-case computational complexity of the basic algorithm is \( O(N^2) \).

**Proof:** We add \( N \) suffixes to the suffix list, and for each suffix we access the list at most \( O(N) \) times. Hence, the aggregate is upper bounded by \( O(N^2) \) accesses. Suppose \( \mathcal{X} = \{0, 1\} \) and \( N = 2k \). Consider a sequence \( x = 1010 \ldots 1001 \) such that \( x_i = 0 \) for \( i \in \{2, 4, \ldots, 2k - 4, 2k - 2, 2k - 1\} \) and \( x_i = 1 \) for \( i \in \{1, 3, \ldots, 2k - 5, 2k - 3, 2k\} \). After having sorted all the suffixes from \( s_{2k+1} \) (the sentinel suffix) down to \( s_{2i} \), where \( i < k \), the suffixes headed by 0 are ordered

\[
(s_{2k-2}, s_{2k-4}, \ldots, s_{2i+2}, s_{2i}, s_{2k-1}),
\]

and the suffixes headed by 1 are ordered

\[
(s_{2k-3}, s_{2k-5}, \ldots, s_{2i+3}, s_{2i+1}, s_{2k}),
\]

so the entire list and intermediate BWT output \( y^{2i-1} \) stacked above it are

\[
\begin{pmatrix}
1 & 1 & \ldots & 1 & \rightarrow & 0 & 0 & 0 & \ldots & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
\text{2k-2} & \text{2k-4} & \ldots & \text{2i+2} & \text{2i} & \text{2k-1} & \text{2k-3} & \text{2k-5} & \ldots & \text{2i+3} & \text{2i+1} & \text{2k} & \text{2k+1}
\end{pmatrix}
\]

When we add \( s_{2i-1} \) to the list, we enter the list at \( s_{2i} \) and try to match \( x_{2i-1} = 1 \) (scanning in the direction of the arrow). The first suffix \( s_{j+1} \) that satisfies \( x_j = 1 \) is \( s_{2k+1} \) (indicated by an asterisk), which is at the end of the list. We had to access the list \( O(k - i) \) times to add \( s_{2i-1} \) to the list. Therefore, in order to add suffixes of the form \( s_{2i-1} \) to the list, for \( i \in \{1, 2, \ldots, k - 1\} \), the aggregate number of accesses required for Case 1 is at least \( O(k^2) \), which is \( O(N^2) \). \( \square \)
5.4 Bidirectional Algorithm

5.4.1 Introduction

The basic algorithm scans for a match by examining previous symbols while scanning up the suffix list. In Section 5.3.5.1, we explained that while running the basic algorithm on a tree source, if a scan for a match to a symbol \( x_i = \alpha \) generated by state \( s \) spills into \( \text{segment}_s \) with \( p(\alpha | \bar{s}) = 0 \), the basic algorithm will need to scan through all of \( \text{segment}_s \). This problem happens only when \( x_i = \alpha \) is the last \( \alpha \) in \( \text{segment}_s \). As \( N \to \infty \), segments will become longer, and the probability that there are additional instances of \( \alpha \) in \( \text{segment}_s \) goes to 1. Therefore, if instead of scanning up the BWT output we scan down as well, asymptotically we will always find another \( \alpha \) in \( \text{segment}_s \).

We thus propose a bidirectional scan through the suffix list that alternates matching attempts in both directions. As explained, this asymptotically eliminates spills, and their associated \( O(|\mathcal{X}|N) \) performance loss in the basic algorithm (viz. (5.9)). A bidirectional scan can thus reduce the typical number of accesses by a factor. No less importantly, it turns out that a bidirectional scan also improves the worst-case performance from \( O(N^2) \) to \( O(N \log(N)) \). This will be proved in Section 5.4.4.

Following Section 5.3.2, define \( \mathcal{S}_i^> \triangleq \mathcal{S}_i = \{s_j : j > i, x_j = x_i, s_j > s_i\} \) and \( \mathcal{S}_i^\leq \triangleq \{s_j : j > i, x_j = x_i, s_j < s_i\} \). These are sets of suffixes already in the list that are larger (\( \mathcal{S}_i^> \)) or smaller (\( \mathcal{S}_i^\leq \)) than \( s_i \), and match it in the first symbol. If \( \mathcal{S}_i^> \) is nonempty, its smallest element can be found, as described in Section 5.3.2. We call this an \textit{up match} for \( x_i \). The new suffix \( s_i \) should be added to the suffix list just before the \( s_j \) that was up-matched. Similarly, if \( \mathcal{S}_i^\leq \) is nonempty, its largest element is the largest suffix \( s_j \) already in the list that satisfies \( s_{j+1} < s_{i+1} \) and \( x_{j} = x_i \). It can be found by scanning down the list (starting from \( s_{i+1} \), and examining decreasing suffixes) until such a \( s_{j+1} \) satisfying \( x_i = x_j \) is found. We call this a \textit{down match} for \( x_i \). The new suffix
$s_i$ should be added to the suffix list just after the $s_j$ that was down-matched. The two matching attempts are performed concurrently. In order to match a symbol in position $l$ in the intermediate BWT output, we scan positions $l-1, l+1, l-2, l+2, \ldots$; the first match, whether a down or an up match, determines where to add $s_i$ to the linked list.

The remainder of the section deals with the computational complexity of the bidirectional algorithm. Conjectures for tree sources are presented in Section 5.4.2, and the typical aggregate number of accesses to the list is bounded in Section 5.4.3. Finally, Section 5.4.4 bounds the worst-case performance of the bidirectional algorithm.

### 5.4.2 Conjectures for tree sources

In Section 5.3.5.1, we conjectured that for irreducible aperiodic tree sources, the typical computational complexity of the basic algorithm is $O(|\mathcal{X}|N)$. If both the upward and downward scans find a match in the initial segment in which the scan began, the analysis for the bidirectional algorithm is identical, and

$$
\sum_{\alpha \in \mathcal{X}} \sum_{s \in S} \sum_{i=1}^{N} \mu_s^\alpha(i) \leq |\mathcal{X}|N. \tag{5.10}
$$

If either the upward or downward scan spill into segments while scanning for a match to $\alpha$ (treated algorithmically either as Case 1 or Case 3), because the bidirectional algorithm also scans in the opposite direction, the expected number of accesses performed in these cases is at worst twice that of the basic algorithm without a spill. Therefore, using arguments similar to those in the discussion following (5.7), the number of accesses caused by spills is $o(|\mathcal{X}|N)$. These observations lead to the following conjecture.

**Conjecture 4** For irreducible aperiodic tree sources, the typical computational complexity of the bidirectional algorithm is at most $O(|\mathcal{X}|N)$. 

153
If there is some \( s \in \mathcal{S} \) and \( \alpha \in \mathcal{X} \) such that \( p(\alpha|s) = 0 \), (5.10) does not hold with equality; hence the expected aggregate number of accesses could be less than \( |\mathcal{X}|N + o(|\mathcal{X}|N) \). This is verified by simulation results in Section 5.7.

### 5.4.3 Typical complexity (binary i.i.d. source)

While a rigorous proof for the conjectured \( O(|\mathcal{X}|N) \) complexity is yet to be found, we can prove a somewhat weaker complexity bound in the special case of a binary i.i.d. source.

**Theorem 20** For binary i.i.d. sources, the expected computational complexity of the bidirectional algorithm is upper bounded by \( O(N \log \log(N)) \).

Because the worst-case result for the hierarchical algorithm presented in Section 5.5 is almost as strong as the expected complexity result of Theorem 20 for the bidirectional algorithm, we only outline the proof of Theorem 20 for brevity.

**Proof (outline):** For a sequence of symbols \( z \in \mathcal{X}^N \), we define a run as a string of consecutive identical symbols, i.e., \( R \triangleq z_i^j, z_i = z_{i+1} = \ldots = z_j \), and \( |R| = j - i + 1 \) as the length of \( R \). Define \( \phi(t) \triangleq t \log(t) - 1 \) for \( t \in \mathbb{R}^+ \). Denote by \( b_k \) the event that a new run begins at position \( k \), i.e., \( i = k \), and denote that run by \( R_k \). We define the function \( \psi \) on the sequence \( z \) as

\[
\psi(z) \triangleq \sum_{k=1}^{\lfloor z \rfloor} 1_{\{b_k(z)\}} \phi(|R_k(z)|),
\]

where \( 1_{\{\cdot\}} \) denotes an indicator function.

The bidirectional algorithm processes the input sequence \( x \) sequentially. After having initialized the list with the sentinel suffix \( s_{N+1} \), and having further processed the \( i \) suffixes \( s_{N} \) down to \( s_{N-i+1} \), \( y^{N-i} \) is a length-\( i \) + 1 permutation of \( x_{N-i}^N \). We define \( \psi_i \triangleq \psi(y^{N-i}) \). For a binary alphabet, it can be proved that the number of accesses to the list performed by the bidirectional algorithm when adding \( s_{N-i} \) to the list is at most \( |\psi_{i+1} - \psi_i| \).
We define the total variation (TV) of $\psi$ as the sum of all the variations of the function over time, i.e.,

$$TV(\psi) \triangleq \sum_{i=0}^{N} |\psi_i - \psi_{i-1}|,$$

where $\psi_{-1} \triangleq 0$. The aggregate number of accesses to the list performed by the bidirectional algorithm is at most $TV(\psi)$. We define $TV_u$ and $TV_d$ as the total variation when $\psi_i$ goes up and down, respectively. It is clear that $TV(\psi) = TV_u(\psi) + TV_d(\psi)$ and $TV_u(\psi) - TV_d(\psi) = \psi_N$. Furthermore, $\psi_N = \psi(y^0)$ where $y^0$ is the final BWT output $y$. Clearly, $\psi_N = -(N+1)$ if and only if no two consecutive symbols in $y$ are identical; hence $\psi_N \geq -(N+1)$, so $TV_d(\psi) \leq TV_u(\psi) + (N+1)$. Therefore, $TV(\psi) \leq 2TV_u(\psi) + (N+1)$.

It thus suffices to show that $TV_u(\psi)$ is “small enough.”

Restricting our attention to binary i.i.d. sources, it can be proved that $E[(\psi_{i+1} - \psi_i)1_{\{\psi_{i+1} > \psi_i\}}] \leq O(\log \log(i))$, leading to a $O(N\log \log(N))$ bound on the expected and typical aggregate number of accesses to the list for the bidirectional algorithm. 

5.4.4 Worst-case complexity

In order to add $s_i$ to the suffix list, a match is found in $y^i$ to $x_i$. This adds $x_{i-1}$ to $y^i$ and creates $y^{i-1}$. We thus consider adding symbols to the intermediate BWT output instead of adding suffixes to the list. We consider constrained intermediate BWT outputs in Lemma 9, and continue by removing the constraints. We upper bound the worst-case aggregate number of accesses to the list required by the bidirectional algorithm by $O\left(|\mathcal{X}|N \log \left(\frac{N}{|\mathcal{X}|}\right)\right)$. An example with $O(N \log(N))$ behavior proves that the bound is tight up to some multiplicative constant.

Define two functions.

$$f(k, l) \triangleq (k - 2)\log(l + 1) + \frac{3}{2} l - \frac{1}{2}, \quad (5.11)$$

$$\eta(k, l, p, q) \triangleq 2 \min\{p, k - p\} + f(p + 1, q) + f(k - p + 1, l - q). \quad (5.12)$$

155
We begin with a technical lemma, proved in Appendix L, that relates $\eta(k,l,p,q)$ to $f(k,l+1)$. This result is then used in the proof of Lemma 9.

**Lemma 8**

$$f(k,l+1) \geq \max_{\{p, q \in \mathbb{N}_0, 0 \leq q \leq l, 1 \leq p \leq k-1\}} \eta(k,l,p,q).$$

**Lemma 9** The aggregate number of accesses to the list required by the bidirectional algorithm, for adding $l$ symbols $\alpha$ to a length-$k$ intermediate BWT output whose first and last symbols are $\alpha$, is upper bounded by $f(k,l)$.

**Proof:** We prove by induction on $l$. For $l = 1$, the BWT output went from $\alpha^{k-2} \alpha_j \alpha^{k-j}$ to $\alpha^k \alpha_j \alpha^{k-j}$, where $\ast$ is used to denote any arbitrary symbol. In the worst case, we need to examine all $k - 2$ arbitrary symbols before matching a symbol $\alpha$, for a total of $k - 1$ accesses. Comparing this to (5.11),

$$f(k,1) = (k - 2) \log(1 + 1) + \frac{3}{2} - \frac{1}{2} = k - 1,$$

and the result holds. Using the inductive hypothesis (5.11), we now need to prove that the upper bound holds for $l + 1$. Consider the first $\alpha$ added to the BWT output, and suppose that it is added after $p$ symbols, i.e., the BWT output went from $\alpha^k \alpha_j \alpha^{k-j}$ to $\alpha^{p-1} \alpha^k \alpha_j \alpha^{k-j}$. We need not consider $p = 0$ because in this case the BWT output becomes $\alpha \alpha^* \cdots \alpha$, which is identical to $p = 1$. Similarly, we need not consider $p = k$, so $1 \leq p \leq k - 1$. In order to match the first $\alpha$ that was added we need at most $\min\{p - 1, k - p - 1\}$ accesses to arbitrary symbols in each direction, another access for the match, and possibly another for an additional arbitrary symbol in the opposite direction, for a total of up to $2 \min\{p, k - p\}$ accesses. We still have $l$ more symbols $\alpha$ to add. Assuming that we need to add $q$ of them in positions before the first $\alpha$ that
was added, and \( l - q \) after it, the inductive hypothesis bounds the aggregate number of accesses further required by \( f((p - 1) + 2, q) + f((k - p - 1) + 2, l - q) \), for a total of at most

\[
2 \min\{p, k - p\} + f(p + 1, q) + f(k - p + 1, l - q) = \eta(k, l, p, q)
\]

aggregate accesses to the suffix list. The proof is completed with Lemma 8.

Lemma 10 removes the constraint that the first and last symbols are \( \alpha \).

**Lemma 10** The aggregate number of accesses to the list required by the bidirectional algorithm, for adding \( l \) symbols \( \alpha \) to a length-\( k \) intermediate BWT output, is upper bounded by \( g(k, l) \), where

\[
g(k, l) \triangleq k \log(l + 1) + \frac{3}{2} l - \frac{1}{2}.
\]  

*Proof:* Denote the length-\( k \) intermediate BWT output at the beginning of the process by \( B_1 \). We construct an alternative length-\( k + 2 \) intermediate BWT output \( B_2 \), such that \( B_2 \) has a symbol \( \alpha_f \) at its first position, and \( \alpha_l \) at its last position, i.e., \( B_2 = \alpha_f B_1 \alpha_l \). The notation \( \alpha_f \) and \( \alpha_l \) stresses that these symbols match \( \alpha \), but the \( l \) symbols \( \alpha \) that need to be added will always come in between them; hence, at each stage in the algorithm, after \( B_1 \) has been turned into \( B_3 \), \( B_2 \) will have been turned into \( \alpha_f B_3 \alpha_l \). We now compare the aggregate number of accesses for adding the \( l \) symbols to \( B_1 \) and to \( B_2 \). For each \( \alpha \) that is added to \( B_2 \), it is in the same position as in \( B_1 \), so there cannot be more accesses required in \( B_1 \). Since the aggregate number of accesses for \( B_2 \) is upper bounded by \( f(k + 2, l) \), we conclude that \( g(k, l) \leq f(k + 2, l) \), and (5.13) is immediate by substituting into (5.11).

Lemma 11 removes the constraint that the \( l \) symbols \( \alpha \) were added after the intermediate BWT output already contained \( k \) symbols.
**Lemma 11** The aggregate number of accesses to the list required by the bidirectional algorithm for matching symbols α, to construct an intermediate BWT output with l symbols α and k non-α symbols, is upper bounded by g(k,l).

**Proof:** The key observation is that, once added to the intermediate BWT output, symbols retain their relative order. Consider two constructions of the same BWT output. In $B_1$ the order in which we add symbols to the BWT output is unconstrained, and in $B_2$ we add the symbols α in the same order as in $B_1$, but only after all the other symbols are already in the BWT output ($B_1$ and $B_2$ here differ from those used in the proof of Lemma 10). Because all symbols have the same relative order, as it is inserted, each symbol α in $B_2$ can match the same α that it matches in $B_1$, but there might be more non-α symbols in between. Therefore, each α that is added cannot require more accesses in $B_1$. Since the aggregate number of accesses for $B_2$ is upper bounded by $g(k,l)$, $B_1$ satisfies this bound also.

Theorem 21 bounds the worst-case aggregate number of accesses to the list by considering all the different symbols that are added to the intermediate BWT output.

**Theorem 21** The aggregate number of accesses to the list required by the bidirectional algorithm to construct a length-$N$ BWT output, with an alphabet of size $|X|$, is upper bounded by $h(N, |X|)$, where

$$h(N, |X|) \triangleq \frac{3}{2}N - \frac{|X|}{2} + N(|X| - 1) \log\left(\frac{N}{|X|} + 1\right).$$  \hspace{1cm} (5.14)

**Proof:** Denote for each $\alpha \in X$ the number of times $\alpha$ will appear in the BWT output by $l_\alpha$; hence

$$\sum_{\alpha \in X} l_\alpha = N.$$  \hspace{1cm} (5.15)
Using Lemma 11, we obtain that the aggregate number of accesses to the list for matching symbols $\alpha$, denoted by $h_\alpha$, satisfies

$$h_\alpha \leq g(N - l_\alpha, l_\alpha). \quad (5.16)$$

By combining (5.13), (5.15), and (5.16), and summing over the alphabet $\mathcal{X}$,

$$h(N, |\mathcal{X}|) = \sum_{\alpha \in \mathcal{X}} h_\alpha$$

$$\leq \sum_{\alpha \in \mathcal{X}} g(N - l_\alpha, l_\alpha)$$

$$= \sum_{\alpha \in \mathcal{X}} \left( (N - l_\alpha) \log(l_\alpha + 1) + \frac{3}{2} l_\alpha - \frac{1}{2} \right)$$

$$= \frac{3}{2} N - \frac{|\mathcal{X}|}{2} + |\mathcal{X}|(N + 1) \log(l_\alpha + 1)$$

$$+ \sum_{\alpha \in \mathcal{X}} (l_\alpha + 1) \log \left( \frac{1}{l_\alpha + 1} \right).$$

The term $h^a$ is the logarithm of the geometric mean of $\{l_\alpha + 1\}_{\alpha \in \mathcal{X}}$, which is maximum when all components are equal. The term $h^b$, up to scaling and an additive constant, is equal to the entropy [1] of the probability mass function $\left\{ \frac{l_\alpha + 1}{N + |\mathcal{X}|} \right\}_{\alpha \in \mathcal{X}}$, and is also maximum when all components are equal. Therefore, $h$ is maximum when $l_\alpha = \frac{N}{|\mathcal{X}|}$, $\forall \alpha \in \mathcal{X}$. The result (5.14) is now straightforward.

Next, we present an alternative and more intuitive derivation of the $O(N \log(N))$ upper bound on the worst-case complexity for a binary alphabet, based on Section 5.4.3. Consider how $TV_\alpha(\psi)$ is maximized. In each iteration $i$, if the longest run has length $K_i$, $TV_\alpha(\psi)$ increases by at most $\phi(K_i + 1) - \phi(K_i)$. But it can be proved that this value is monotone increasing in $K_i$; thus, the longer the longest run, the more we can increase
$TV_u(\psi)$. We conclude that $TV_u(\psi)$ is maximized by growing one long run, so

$$
TV(\psi) \leq 2TV_u(\psi) + (N + 1)
< 2\phi(N) + (N + 1)
= O(N\log(N)). \tag{5.17}
$$

Recalling that $TV(\psi)$ upper bounds the aggregate number of accesses to the suffix list, (5.17) is an upper bound on the worst-case complexity.

Next, by providing an example containing a length-$O(N)$ run, we prove that the worst-case is indeed $O(N\log(N))$.

**Theorem 22** For a binary alphabet, the worst-case aggregate number of accesses to the list required by the bidirectional algorithm to construct a length-$N$ BWT output is $O(N\log(N))$.

**Proof:** Because Theorem 21 provides an upper bound, it suffices to prove the existence of a length-$N$ input sequence that requires $O(N\log(N))$ aggregate accesses to the list while running the bidirectional algorithm on it. In the proof, we assume that our algorithm processes the symbols in regular order $x_1$ up to $x_N$, instead of reversed order, because this simplifies the presentation. Hence, the terminology will use prefixes and following symbols, instead of suffixes and preceding symbols, respectively. The proof is broken up into three stages. We first provide a source that generates a length-$\frac{3}{4}N$ sequence $x^1$ whose intermediate BWT has a long run (its expected length is $\frac{1}{4}N + o(N)$). Then we explain how to construct a sequence $x^2$ that, appended to $x^1$, requires an expected $O(N\log(N))$ aggregate accesses to the suffix list for breaking up the long run. Finally, we explain why the construction of $x^1$ and $x^2$ proves the required result.

**First stage** - consider the Markov source in Figure 5.4. Each time it is in state $s0$, it either transmits 0, or 10 by going through state $s1$, each with probability 0.5. We let
Figure 5.4 A Markov source that at each step either transmits the symbol 0 or the two symbols 10, each with probability 0.5. (We label state transitions with the transition probability followed by the output).

the source generate $x^1$ with $\frac{3}{4}N$ symbols. The expected number of symbols generated per visit to $s0$ is $\frac{3}{2}$, and the expected number of times 1 appears in $x^1$ is $\frac{1}{4}N + o(N)$. After running the bidirectional algorithm on $x^1$, all the prefixes whose last symbol is 1 are grouped together in the list, and since the structure of the source prevents two consecutive appearances of 1 in $x^1$, the symbols in segment 1 of the intermediate BWT output are all 0. These symbols form a long run in the intermediate BWT output. The expected number of times 1 appears in $x^1$ is $\frac{1}{4}N + o(N)$, so the expected length of the long run is $\frac{1}{4}N + o(N)$.

Second stage - the entropy of the source in Figure 5.4 is $\frac{2}{3}$ bits per symbol. Therefore, on average, two adjacent prefixes in the list, $p_i$ and $p_{next_i}$, share their last $\frac{3}{2} \log(N) + O(1)$ symbols (the preceding symbol of $p_i$ is smaller than the preceding symbol of $p_{next_i}$; “preceding” is w.r.t. the shared parts). Suppose we want to place a new prefix $\tilde{p}$ between them; first we generate $O(1)$ symbols that would place $\tilde{p}$ in between the preceding symbols of the shared parts, then we copy the shared parts. The average length of $\tilde{p}$ is $\frac{3}{2} \log(N) + O(1)$, and it does not disrupt the structure of the source. After constructing $x^1$, we generate a $\tilde{p}$ that is in the middle of the long run. Instead of then outputting 0, as the source dictates, we output 1. The bidirectional algorithm will scan the entire long run, requiring $\frac{1}{4}N + o(N)$ expected accesses. Now, instead of one long length-$\frac{1}{4}N + o(N)$ run, we have two length-$\frac{1}{8}N + o(N)$ runs (expected lengths). Just as we split the long

161
run in two, we split each of the length-$\frac{1}{8}N + o(N)$ runs into two length-$\frac{1}{16}N + o(N)$ runs. This requires another $\frac{1}{4}N + o(N)$ expected accesses. Continuing in this manner, in iteration $i \in \{1, \ldots, \lceil \frac{1}{2} \log(N) \rceil \}$ we split $2^{i-1}$ runs, each of expected length $2^{-i-1}N + o(2^{-i}N)$ (we neglect additional accesses required for the prefixes that we constructed during this procedure). This requires at least $(\frac{1}{2} \log(N)) \left( \frac{1}{4}N + o(N) \right) = O(N \log(N))$ expected aggregate accesses. We generate $x^2$ by concatenating all the prefixes that we generated and all the symbols 1 that were used for splitting runs. In total we had to split $\sum_{i=1}^{\lceil \frac{1}{2} \log(N) \rceil} 2^{i-1} = O(\sqrt{N})$ runs, each requiring $\frac{3}{2} \log(N) + O(1)$ expected symbols, so $\mathbb{E}||x^2|| = O\left( \sqrt{N} \log(N) \right)$.

**Third stage** - $\mathbb{E}[|x^1| + |x^2|] = \frac{3}{2}N + O\left( \sqrt{N} \log(N) \right)$; hence $\mathbb{E}[|x^1| + |x^2|] < N$ asymptotically, and the expected aggregate number of accesses to the suffix list is $O(N \log(N))$. This is true in expectation, so there must exist a length-$N$ input sequence that requires $O(N \log(N))$ aggregate accesses while running the bidirectional algorithm on it.

\[\square\]

### 5.5 Hierarchical Lists

Consider enhancing the bidirectional algorithm by using two hierarchically related suffix lists, such that the lower list contains all the suffixes, and the upper list is a sparse version of the lower list. Each parent node in the upper list corresponds to approximately $K$ (between $K$ and $2K - 1$) consecutive suffixes in the lower list; those suffixes are called a block. We also refer to the portion of the intermediate BWT output corresponding to a block of suffixes as a block. If possible, a match is made inside the block, else we scan for a match using the upper list. By maintaining in each parent node flags indicating what symbols are in its block, instead of scanning the entire block for a match in the lower list, a single access to the flag suffices, thus speeding up the scan.
We now describe a simple implementation (advanced implementations might use hash tables to increase speed and save memory).

- A parent node has a counter indicating the size of the block, a pointer to the smallest suffix in the block, and flags indicating what symbols are in the block. The size of the blocks is maintained between $K$ and $2K - 1$ by splitting a block into two size-$K$ blocks each time its size is about to become $2K$. This requires at most $\frac{N}{K}$ nodes in the upper list.

- The lower list can be either a singly linked suffix list (only next pointers) or a doubly linked suffix list. In singly linked lists, the previous suffix can be found by first using the parent node to reach the start of the block, and then scanning up, with a computational complexity of $O(K)$ at most. The upper list is doubly linked and is scanned bidirectionally for improved performance.

- We can access the parent node from its block by adding a pointer from every suffix to its parent. Alternatively, a suffix can point to either another suffix or its parent, but this requires adding one indicator bit per suffix.

The data structure of hierarchical lists appears in Figure 5.5. Using singly linked suffix lists with indicator bits requires fewer pointers, hence less memory.

On current typical computer architectures, we need 5 bytes (1 for the symbol itself and 4 for a pointer) per suffix for a singly linked lower list, and 9 bytes (1 for the symbol itself and 8 for 2 pointers) for a doubly linked lower list. Pointers from each suffix to its parent use 4 bytes per suffix, while indicator bits use $\frac{1}{8}$ byte per suffix. Nodes in the upper list use 8 bytes for pointers to adjacent nodes in the doubly linked upper list, 4 bytes for a pointer to the smallest suffix in the block, $\frac{|K|}{8}$ bytes for flags, and 1 byte for a counter (we will later see why we assume $2K \leq 256$). As explained previously, the
Figure 5.5 The structure of hierarchical lists. (Pointers inside the same list are solid lines, and pointers in between hierarchies are dashed.)

number of nodes in the upper list is at most $\frac{N}{K}$. The total memory use is summarized in Table 5.1. A singly linked lower list with indicator bits will be most memory-efficient, but will be slowest because backwards movement in the lower list and access to parent nodes will both be computationally expensive. A doubly linked lower list with pointers to parents will use the most memory, but will be fastest.

Table 5.1 The memory use of hierarchical lists (in bytes) on current typical computer architectures. (The basic and bidirectional algorithms both use 9 bytes per suffix.)

<table>
<thead>
<tr>
<th>Memory</th>
<th>Hierarchical implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower list</td>
</tr>
<tr>
<td></td>
<td>Singly linked</td>
</tr>
<tr>
<td>Pointing to parents</td>
<td>Pointers</td>
</tr>
<tr>
<td>Memory per suffix</td>
<td>$5 + 4</td>
</tr>
<tr>
<td>Parent node memory</td>
<td>$8 + 4</td>
</tr>
<tr>
<td>Total per suffix</td>
<td>$9 + \frac{13+\frac{1}{K}}{K}$</td>
</tr>
</tbody>
</table>

164
In terms of computation, every suffix added requires at most $O(K)$ accesses to the lower list, hence an aggregate of $O(KN)$. Furthermore, since there are at most $\frac{N}{K}$ blocks, and each block split is $O(K) + O(|\mathcal{X}|)$, the computational effort expended on splitting blocks is at most $O(N) + O\left(\frac{|\mathcal{X}|N}{K}\right)$. For a scan that requires $a$ accesses in the bidirectional algorithm, we need $O\left(\frac{a}{K}\right)$ accesses to the upper list. Since the aggregate number of accesses performed by the bidirectional algorithm was bounded by $O\left(|\mathcal{X}|N \log\left(\frac{N}{|\mathcal{X}|}\right)\right)$, in the worst case we need $O\left(\frac{|\mathcal{X}|N}{K} \log\left(\frac{N}{|\mathcal{X}|}\right)\right)$ aggregate accesses to the upper list. The total computational complexity is thus $O\left(N \left[\frac{|\mathcal{X}|}{K} \log\left(\frac{N}{|\mathcal{X}|}\right) + K\right]\right)$, and the optimal $K$ is $O\left(\sqrt{|\mathcal{X}| \log\left(\frac{N}{|\mathcal{X}|}\right)}\right)$. Summarizing this discussion, we have the following result.

**Theorem 23** The worst-case total aggregate number of accesses to both lists required by the hierarchical algorithm to construct a length-$N$ BWT output, with an alphabet of size $|\mathcal{X}|$, is upper bounded by $O\left(N \sqrt{|\mathcal{X}| \log\left(\frac{N}{|\mathcal{X}|}\right)}\right)$.

After providing a worst-case result, we now discuss the typical case. If indicator bits or a singly linked lower list are used, the expected number of accesses to the lower list will be $O(K)$ per suffix, and if $K = O\left(\sqrt{|\mathcal{X}| \log\left(\frac{N}{|\mathcal{X}|}\right)}\right)$, the number of accesses to the lower list will be $O\left(N \sqrt{|\mathcal{X}| \log\left(\frac{N}{|\mathcal{X}|}\right)}\right)$. Therefore, with this $K$, the worst-case bound of Theorem 23 also represents the typical case. Alternatively, when a doubly linked lower list with pointers to parents is used, most suffixes that we process will require only a few accesses to the lower list, and the typical complexity will be better than the worst-case performance. Simulation results in Section 5.7 confirm that the total aggregate number of accesses performed by the hierarchical algorithm to both lists is often low.

**Block length:** Previously, we compared $O(KN)$ accesses to the lower list with $O\left(\frac{|\mathcal{X}|N}{K} \log\left(\frac{N}{|\mathcal{X}|}\right)\right)$ accesses to the upper list. Although these are $O(\cdot)$ terms, they both deal with accesses to lists, and it is reasonable to assume that the constants in the two expressions are similar, so we expect $K \sim \sqrt{|\mathcal{X}| \log\left(\frac{N}{|\mathcal{X}|}\right)}$. Furthermore, we might prefer
a hierarchical algorithm that is better on average, and slightly worse in the worst case. This would result in a bias towards smaller $K$. According to this discussion, $K$ should be quite small. This will be verified by simulation in Section 5.7.

**Hierarchical vs. bidirectional:** Recall that the worst-case complexity of the hierarchical algorithm is $O(N\sqrt{\log(N)})$, while the worst-case complexity of the bidirectional algorithm is $O(N\log(N))$. Furthermore, Theorem 20 provided a $O(N\log\log(N))$ upper bound on the expected aggregate number of accesses to the list performed by the bidirectional algorithm on binary i.i.d. inputs. This comparison shows that the hierarchical algorithm should be better than the bidirectional algorithm asymptotically. Simulation results in Section 5.7 indicate that the hierarchical algorithm often outperforms the bidirectional algorithm in practice, except when the alphabet is small.

Finally, the hierarchical structure suggests further enhancements. First, we can construct a tree structure that points to suffixes; if a match is not quickly found while scanning the BWT output, the suffix is matched in $O(\log(N))$ accesses to the tree. This provides even faster worst-case behavior. Another algorithm that the hierarchical structure suggests is using a singly linked suffix list with a supplemental sparse set of backwards pointers. This leads to a memory-efficient algorithm using slightly above 5 bytes per suffix.

### 5.6 Further Enhancements and Applications

One approach to enhance the core of the algorithm was studied in Sections 5.4 and 5.5, where we presented the bidirectional and hierarchical algorithms. Another enhancement is suggested in Section 5.6.1. Instead of running suffix sorting on the original input sequence, we use a reduced alphabet by partitioning the (original) input symbols. In some cases, reduced alphabets accelerate our algorithms substantially.
In comparison with other suffix sorting methods (see Section 5.2.2), suffix lists do not have the best worst-case complexity, nor are they the fastest in practice (see Section 5.7). Rather, the distinguishing feature of suffix lists is the simplicity of the core of the algorithm. Another research direction that exploits this simplicity is described in Section 5.6.2, where we show how the bidirectional algorithm leads to a simple VLSI implementation. Finally, Section 5.6.3 discusses the potential application to a high speed BWT-based compression system in VLSI. Before continuing, we point out that the objective of these discussions is to emphasize trade-offs between different possibilities, without going into great detail.

5.6.1 Reduced alphabet

Using Conjectures 3 and 4, for typical input sequences the computational complexity of the basic and bidirectional algorithms is at most $O(|\mathcal{X}|N)$. Therefore, a substantial speedup can be achieved by using small alphabets. We convert each symbol of $x$ into a string of several symbols from a smaller alphabet $\tilde{\mathcal{X}}$, generating a longer sequence $\tilde{x}$. We then run suffix sorting on $\tilde{x}$. It turns out that we can still obtain the order of the suffixes of $x$. In particular, if $|\mathcal{X}| = k^l$ and $|\tilde{\mathcal{X}}| = k$, $\mathcal{X}$ can be partitioned into $l$ symbols, and the suffix $\tilde{s}_{j+1}$ of $\tilde{x}$ corresponds to $s_{j+1}$ of $x$. This method is described in detail by Effros et al. [25]. While the memory use increases linearly in $l$, the computation goes from $O(Nk^l)$ to $O(Nkl)$. However, there is overhead in converting between alphabets. Therefore, we should not reduce the alphabet too much. Some simulation results obtained with reduced alphabets appear in Section 5.7.

More elaborate forms of alphabet reduction are also possible. For example, Huffman coding [1] of the individual symbols of the original input sequence $x$ can be used to reduce the alphabet size with a smaller concomitant increase in the length of the reduced alphabet sequence $\tilde{x}$ than with the straightforward $l$-ary coding method described earlier.
Figure 5.6 Block diagram of a proposed VLSI implementation of the bidirectional algorithm.

For brevity, we do not pursue this here. We conclude that reduced alphabets can be used to accelerate our algorithms, but the increased memory use makes reduced alphabets less attractive.

5.6.2 VLSI implementation of the bidirectional algorithm

The bidirectional algorithm leads to a fast VLSI implementation. Figure 5.6 contains a block diagram of such an implementation. The up scan of the BWT output is performed by state machine SM1 that accesses RAM1 to fetch the symbol preceding the suffix and the next pointer. Similarly, SM2 and RAM2 perform the down scan. Once SM1 or SM2 find a match they raise the done flag. After SM3, the main state machine, detects that a done signal has been raised, it updates the contents of RAM1 and RAM2 using the we (write enable) and d (data) signals. SM3 also controls two multiplexers that address the RAMs either from SM3 or from the secondary state machines.

RAM1 and RAM2 store the next and previous pointers, respectively, as required by the bidirectional algorithm. We also store the x sequence in both RAMs because during the scan for a match both SM1 and SM2 access the RAMs every clock cycle. This
redundancy is reasonable, considering that SM1 and SM2 run the up and down scans in parallel, speeding up the algorithm substantially. Alternatively, a slower implementation could store $x$ once and interleave between up scans and down scans.

For an implementation that runs on $N$-bit blocks, where $k$-bit symbols are used, each pointer has $\lceil \log(N/k) \rceil$ bits. There are $N/k$ symbols, and for each of these the symbol itself is stored twice, and there are two pointers. The total memory $M$ (in bits) used by the RAMs is thus

$$M \approx \left( \frac{N}{k} \right) 2 \left( k + \lceil \log(N/k) \rceil \right)$$

$$= 2N \left( 1 + \frac{\lceil \log(N/k) \rceil}{k} \right),$$

so using larger symbols is more efficient. On the other hand, large symbols increase the computation substantially because $|\mathcal{X}| = 2^k$. Denoting the time required for the computation by $T$, and using Conjecture 4, we have

$$T = O \left( \frac{N}{k} 2^k \right)$$

for typical inputs. The result is a trade-off between memory use and computation.

### 5.6.3 BWT-based compression systems in VLSI

Consider using the bidirectional algorithm as proposed in Figure 5.6 to compute the BWT output on a length-$N = 10^6$ input partitioned into $k = 4$ bit symbols. The memory use will be

$$M \approx 2 \cdot 10^6 \left( 1 + 18/4 \right) = 11 \cdot 10^6$$

bits, which seems to be at or near the range of current VLSI technology. We can also estimate the speed of these chips. Assuming that we perform one iteration for both the up and down matches every clock cycle, and that the total number of iterations spent searching for matches is $|\mathcal{X}|N$, with $250 \cdot 10^3$ symbols over a size-$|\mathcal{X}| = 2^k = 16$ alphabet
we need \((\frac{1}{2})\(250 \cdot 10^3\) \(16 = 2 \cdot 10^6\) clocks for finding matches. It also seems reasonable to assume an overhead of 10 additional clocks per input symbol, thus requiring \(2.5 \cdot 10^6\) clocks of overhead. Finally, for finding matches and for overhead we need a total of \(4.5 \cdot 10^6\) clocks, or 4.5 clocks per input bit. Rates of 100 Mbps seem at or near the range of current VLSI technology. Actually, as the simulation results of Section 5.7 indicate that the bidirectional algorithm often requires much less than \(|\mathcal{X}|N\) aggregate accesses in practice, performance might be even better.

A traditional BWT-based compression system appears in Figure 5.1. Such a system might include a BWT block and a PIID compression block in the encoder, a PIID decompression block and an inverse BWT block in the decoder. To implement these in VLSI, we would begin with the BWT building block described in Section 5.6.2. Easily implementable methods for compressing and decompressing PIID blocks can be selected from the range of such methods [21–23, 25, 26]. Finally, the inverse BWT [21, 25, 26, 31] is simple, and we feel that it can also be implemented easily in VLSI.

Alternatively, one could attempt to implement BWT-MDL (see Chapter 2). Assuming a binary alphabet, one of the difficult computational feats in BWT-MDL involves partitioning a \(segment_s\) into two constituent segments, \(segment_{0s}\) and \(segment_{1s}\). Then, BWT-MDL accumulates statistical information (counts the number of appearances of 0 and 1) on each of these segments. Solving the partitioning of each segment with a binary search has a superlinear worst-case complexity. In Chapter 2 we overcame this super-linear worst-case complexity by using (compact) prefix trees. An alternative approach, which we leave for future research, involves hardware support for the binary searches, and for accumulating statistical information on the BWT output.
5.7 Simulation Results

5.7.1 Preliminaries

We ran suffix sorting on several files and compared our method to several others. We also compared different alphabet sizes and studied how the block length \( K \) affects the performance of the hierarchical algorithm. We used a SUN Ultra-Enterprise, with a 336-MHz SUNW UltraSPARC-II CPU and 2GB memory, using the SunOS 5.8 operating system. Programs were compiled with gcc 2.95.2 with option -O3 for optimization. Times reported were measured with the user time returned by the `rusage` command, and averaging over 20 runs. The files on which we ran suffix sorting were taken from the Calgary and Canterbury corpora (http://corpus.canterbury.ac.nz). Table 5.2 describes these files, their sizes, and their average matching length (AML).

\[ \text{Table 5.2 Files used for simulation. (File sizes are in bytes.)} \]

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>AML</th>
<th>Corpus</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pic</td>
<td>(0.51 \cdot 10^6)</td>
<td>2837.07</td>
<td>Calgary</td>
<td>image</td>
</tr>
<tr>
<td>book1</td>
<td>(0.77 \cdot 10^6)</td>
<td>7.32</td>
<td>Calgary</td>
<td>text</td>
</tr>
<tr>
<td>world192.txt</td>
<td>(2.47 \cdot 10^6)</td>
<td>19.35</td>
<td>Canterbury</td>
<td>text</td>
</tr>
<tr>
<td>bible.txt</td>
<td>(4.05 \cdot 10^6)</td>
<td>12.64</td>
<td>Canterbury</td>
<td>text</td>
</tr>
<tr>
<td>E.coli</td>
<td>(4.64 \cdot 10^6)</td>
<td>12.99</td>
<td>Canterbury</td>
<td>DNA sequence</td>
</tr>
</tbody>
</table>

We implemented the basic, bidirectional, and hierarchical algorithms in software. Using Approximation 2, when our algorithms scan the list for a match, the location in memory of the next list element is random. Hence, cache misses dominated the performance. In order to improve performance, we reduced the number of cache misses by aligning list elements along cache line boundaries. On current typical computer architectures, cache lines are \(2^k\) bytes long. We changed the sizes of our list elements accordingly, instead of the values in Table 5.1. List elements of the basic and bidirectional algorithms

\[ \text{4The AML was calculated with the suffix sorting software provided by J. Seward.} \]
contain two pointers and a character. Instead of using 9 bytes, we packed a 24 bit index\textsuperscript{5} and 8 bit character into a 4 byte integer, for a total of 8 bytes per suffix. We implemented the hierarchical algorithm with pointers to parents and a doubly linked lower list; lower list elements thus require three pointers and a character. Instead of using 13 bytes, we promoted the character to a 4 byte integer, for a total of 16 bytes per suffix. Similarly, we implemented upper list elements with 48 bytes instead of 45.

5.7.2 Alphabet size

Table 5.3 shows that, for the text file *book1*,\textsuperscript{6} reduced alphabets decrease the number of accesses to the suffix list performed by the basic and bidirectional algorithms. The runtime is affected accordingly. For *book1*, the first row shows results with a full alphabet, where $|\mathcal{X}|$ is the number of different ASCII characters that appear in the file. The following rows are for reduced alphabets. The *E.coli* file is a sequence of DNA symbols with four symbols (A, C, G, and T), thus $|\mathcal{X}| = 4$, and there is no point in reducing the alphabet further. Table 5.3 also includes results for the hierarchical algorithm using the original alphabet, with a block size of $K = 20$. Because the number of accesses to both lists was much smaller, there was no point in reducing the alphabet.

For the basic algorithm, the aggregate number of accesses was similar to $|\mathcal{X}|N$. Although the number of accesses to the list for $|\mathcal{X}| = 4$ and $|\mathcal{X}| = 2$ were close, the binary alphabet was usually slower because of the overhead when dealing with longer sequences. The bidirectional algorithm required less than $|\mathcal{X}|N$ accesses to the suffix list, as discussed in Section 5.4.2. For the hierarchical algorithm, the total aggregate number of accesses to both lists was between $3N$ and $7N$ for all the files we tested.

Table 5.3 also shows how many of the accesses to the list were for Case 3; this indicates the performance penalty due to spills. For text files with the original alphabet, the basic

\textsuperscript{5}Using 24 bit indices decreases the range of possible input sizes.

\textsuperscript{6}The same behavior holds for all the large-alphabet files that we tested.
Table 5.3 Total aggregate accesses to the list, and aggregate accesses for Case 3, performed by the basic, bidirectional, and hierarchical algorithms, for different alphabet sizes. (Run times are in seconds.)

| Algorithm   | File | $N$   | $|\mathcal{X}|$ | Time | Case 3 | Total | $|\mathcal{X}|N$ |
|------------|------|-------|----------------|------|--------|-------|--------------|
| Basic      | book1| 0.77·$10^6$ | 82  | 11.2 | 23.20·$10^6$ | 67.22·$10^6$ | 63.04·$10^6$ |
|            |      | 1.54·$10^6$ | 16  | 5.9  | 2.62·$10^6$  | 24.25·$10^6$ | 24.60·$10^6$ |
|            |      | 3.08·$10^6$ | 4   | 4.1  | 8112   | 13.29·$10^6$ | 12.30·$10^6$ |
|            |      | 6.15·$10^6$ | 2   | 4.4  | 662   | 13.17·$10^6$ | 12.30·$10^6$ |
| E.coli     |      | 4.64·$10^6$ | 4   | 7.2  | 149   | 18.57·$10^6$ | 18.55·$10^6$ |
| Bidirectional | book1| 0.77·$10^6$ | 82  | 4.0  | 1.12·$10^6$ | 23.65·$10^6$ | 63.04·$10^6$ |
|            |      | 1.54·$10^6$ | 16  | 2.4  | 7252  | 9.90·$10^6$  | 24.60·$10^6$ |
|            |      | 3.08·$10^6$ | 4   | 2.2  | 1300  | 6.77·$10^6$  | 12.30·$10^6$ |
|            |      | 6.15·$10^6$ | 2   | 3.0  | 12    | 8.71·$10^6$  | 12.30·$10^6$ |
| E.coli     |      | 4.64·$10^6$ | 4   | 6.4  | 107   | 16.43·$10^6$ | 18.55·$10^6$ |
| Hierarchical | book1| 0.77·$10^6$ | 82  | 1.7  | 0.03·$10^6$ | 4.69·$10^6$  | 63.04·$10^6$ |
|            |      | 4.64·$10^6$ | 4   | 11.9 | 97    | 16.35·$10^6$ | 18.55·$10^6$ |

algorithm often expended 30 – 40% of the computation on Case 3. In these cases, the bidirectional algorithm improves the performance substantially because it often avoids spills by scanning in both directions. With the hierarchical algorithm, spills become even less important because the upper list accelerates long scans for matches. Finally, we note that the computational resources expended on spills can also be reduced by using reduced alphabets.

5.7.3 Block lengths for the hierarchical algorithm

We implemented the hierarchical algorithm with pointers to parents, using a doubly linked lower list. Because access to a parent node required one operation, and the large majority of matches were in the same block, the total number of accesses to both lists were in the range between 3$N$ and 7$N$ for all the files that we tested. This demonstrates the strength of the hierarchical algorithm - fast access in the lower list when a match is near, and accelerated access in the upper list as a fallback. Furthermore, because the
Figure 5.7 The effect of the block length \( K \) on the performance of the hierarchical algorithm. (Run times are in seconds.)

overhead in maintaining two suffix lists is substantial, and the number of accesses was low to begin with, using reduced alphabets always degraded the performance.

In Section 5.5, we discussed how the block length affects the performance of the hierarchical algorithm. Figure 5.7 plots the run times for the files listed in Table 5.2, for \( K \) between 1 and 100. For all the files except the DNA file, the optimal \( K \) was between 15 and 25. This reflects a tradeoff between a small number of accesses to the lower list (small \( K \)) and accelerated scans in the upper list (large \( K \)). For the DNA file, because of the small alphabet, the upper list was hardly beneficial, while using larger \( K \) decreased the computation required for splitting blocks.
Table 5.4 Run times for the basic (B), bidirectional (BD), and hierarchical (H) algorithms; suffix trees by Kurtz (K); doubling method by Larsson-Sadakane (LS); ternary-split by Seward (S). (Run times are in seconds.)

<table>
<thead>
<tr>
<th>File</th>
<th>Sorting method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
</tr>
<tr>
<td>pic</td>
<td>1.9</td>
</tr>
<tr>
<td>book1</td>
<td>4.1</td>
</tr>
<tr>
<td>world192.txt</td>
<td>16.1</td>
</tr>
<tr>
<td>bible.txt</td>
<td>29.8</td>
</tr>
<tr>
<td>E.coli</td>
<td>7.2</td>
</tr>
</tbody>
</table>

5.7.4 Comparison to other methods

In order to compare our method to other suffix sorting methods, for the basic and bidirectional algorithms we decided to use a reduced alphabet with $|\mathcal{X}| = 4$, except for the DNA file, where there were only four symbols in the file to begin with. The hierarchical algorithm was used without alphabet reduction and with $K = 20$. We compare our basic, bidirectional, and hierarchical algorithms to the suffix tree method of Kurtz [32], the improved doubling method of Larsson and Sadakane [34], and the ternary-split method as implemented by Seward [35]. Run times appear in Table 5.4.

Among our algorithms, the bidirectional algorithm outperforms the basic algorithm on every file. The hierarchical algorithm is even faster, except for the DNA file. Because that file has a small alphabet, the number of accesses to the list performed by the basic and bidirectional algorithms was already small, while the hierarchical algorithm requires maintaining a more complicated data structure.

The hierarchical algorithm, whose worst-case complexity is $O(N\sqrt{\log(N)})$, was faster than Kurtz's linear complexity suffix tree construction [32] on every file. On most files, the hierarchical algorithm is faster than the improved doubling method of Larsson and
Sadakane [34], whose worst-case complexity is $O(N \log(N))$. However, Seward’s improved implementation of the ternary-split method [35] is fastest by a large factor, and is only beaten on the DNA file by the bidirectional algorithm. Unfortunately, Seward’s method is not only nonsequential, but also has $O(N^2)$ worst-case complexity. Although these run times only reflect the performance of specific software implementations with the compiler and computer that we used for simulation, they are nonetheless useful for getting a rough impression of the comparative performance.

Another dimension that needs to be considered when comparing the different methods is their memory use. Because the basic and bidirectional algorithms were used with a reduced alphabet on all but the DNA file, they require 36 bytes per input symbol (in the original alphabet) on current typical computer architectures. Using Table 5.1, an implementation of the hierarchical algorithm that uses a doubly linked lower list with pointers to parents and $K = 20$ requires approximately 15 bytes per input symbol. (Taking the cache line size into account, our implementations used approximately 32 and 18 bytes per input symbol, respectively; see Section 5.7.1 for details.) Therefore, in practice the hierarchical algorithm outperforms the basic and bidirectional algorithms in both speed and memory use. The version of the hierarchical algorithm that we implemented is fastest, but consumes the most memory (see Table 5.1). Therefore, our implementation, while consuming more memory than the other suffix sorting methods that we simulated (see Section 5.2.2), can be improved in this respect.

5.8 Conclusions and Future Work

By investigating the ordering of suffixes in a suffix list data structure, we established many useful properties and came up with the core of our algorithm, which determines where to insert a new suffix by scanning the intermediate BWT output for the symbol
that heads it. The basic algorithm, which scans the BWT in one direction, has worst-case computational complexity of $O(N^2)$. By scanning the BWT output in both directions, the bidirectional algorithm has worst-case computational complexity of $O(\left|\mathcal{X}\right| \cdot N \log \left(\frac{N}{|\mathcal{X}|}\right))$. By maintaining two hierarchies of lists, the worst-case complexity is further reduced to $O\left(N \sqrt{|\mathcal{X}| \log \left(\frac{N}{|\mathcal{X}|}\right)}\right)$.

The algorithms can be accelerated by reducing the alphabet, but this uses more memory. The hierarchical algorithm also presents various tradeoffs between speed and memory efficiency. Simulation results indicate that our algorithms have run times that are roughly linear in the size of the file, and are competitive with other suffix sorting methods, especially for inputs over small alphabets such as DNA.

However, the distinguishing feature of our suffix lists method is neither the worst-case complexity nor their competitive speed in practice. Rather, the simplicity of the method suggests many refinements to the basic algorithm and leads to a proposed VLSI implementation of the bidirectional algorithm. The potential application to a high speed BWT-based compression system in VLSI seems promising.

Future work can concentrate on VLSI implementations for these algorithms. One would need to pay close attention to the bottlenecks in such an implementation, in particular the memory system, and try to optimize their design. Another direction in VLSI implementation is the study of algorithms that retain the simplicity of suffix lists and help accumulate statistical information on the BWT output, as required by BWT-MDL.

Finally, a more theoretically oriented topic for future work involves the development of compression algorithms that, instead of relying on suffix lists as a computational tool, use its sequential nature to their advantage. With suffix lists, the BWT output is constructed sequentially, hence when adding a new symbol to the BWT output, its neighboring symbols reside in the same segment. Adopting a tree source model, it makes sense to
estimate the conditional probability in the segment according to the neighboring symbols. In particular, the number of accesses required until a match is found can be formulated into a coarse probability estimate. This approach could lead to compression methods that are implemented on top of our algorithms, and whose compression performance is related to the speed of suffix lists.
CHAPTER 6

DISCUSSION AND FUTURE WORK

6.1 Discussion

To the best of our knowledge, BWT-MDL is the first $O(N)$ implementation of an (unbounded depth) semipredictive encoder. Our results on two-part codes indicate that they can be used for universal compression in a distributed framework. The parallel compression algorithm leverages the results of both contributions. A work-efficient parallel source coding algorithm that compresses almost as well as the best serial algorithms was developed by merging the context tree pruning aspect of BWT-MDL with the distributed potential of two-part codes.

Suffix lists can be viewed as a simple, fast, and low-memory method for suffix sorting. However, this perspective does not do them justice. The distinguishing feature of suffix list algorithms is that they are simple enough to be implemented in hardware. The BWT, which underlies some of the most efficient high-performance universal compression algorithms to-date [21, 24, 25], as well as our BWT-MDL algorithm, is often their computational bottleneck. Hardware implementations of the BWT via suffix list algorithms open up a new range of potential applications for fast high-performance compression.
6.2 Unbounded Depth Parallel Algorithms

We envision a more sophisticated parallel compression algorithm than the one we discussed. The advanced algorithm will support unbounded context depths, which will provide low redundancy performance over a much broader class of sources. By partitioning the input into blocks and processing the blocks in parallel, parallel unbounded depth algorithms will be able to support huge data rates, and may lead to new applications that until now were limited by the throughput bottleneck of serial compression algorithms.

6.3 Directions for Future Work

Many directions for future work were pointed out in the previous chapters. Instead of repeating those points here, we describe the possible directions in a broader perspective. The main research direction involves the development of a parallel algorithm that will support unbounded context depths. This will enable good compression performance over a broader class of tree sources. The actual implementation of such a system in hardware will be a major endeavor. Additional aspects of parallel unbounded depth algorithms that require more work include heuristics to improve the compression performance in practice, and a possible application to lossy compression.

Another direction for research is an application of our algorithms to lossless image compression. Yet another direction is based on the relations between universal compression, classification, channel decoding, and other problems in statistical inference. An efficient universal source coding algorithm may lead to efficient solutions for additional related problems, such as genome analysis.

Finally, our work-efficient parallel source coding algorithm is just one option in the design space for parallel compression algorithms that was laid out in Chapter 4. Future research could attempt to further map out that space, and provide additional (parallel)
algorithms that offer different tradeoffs between throughput, redundancy, delay, memory use, and power consumption. The $O(N/B)$ parallel algorithm pays a price of roughly $B \log(N)$ bits for a factor-$B$ speedup. This can be viewed as an upper bound on the extra redundancy required for parallelization. Alternatively, we can overcome this by overlapping blocks in the encoder at the cost of a serial algorithm in the decoder. However, it is not clear whether there is an inherent need for this extra redundancy. In order to conduct future research on parallel source coding in fruitful directions, it will be crucial to undertake a fundamental study of redundancy bounds for algorithms lying in different regions of the parallel design space.
APPENDIX A

CONTEXT TREE PRUNING AS DYNAMIC PROGRAMMING

We begin with a brief review of dynamic programming (DP) in Section A.1. We then formulate Context Tree Pruning (CTP) in Section A.2, and express the coding length such that it leads to a correspondence between CTP and DP. Some additional notation is defined in Section A.3, and the correspondence between DP and CTP is provided in Section A.4.

A.1 Review of Dynamic Programming (DP)

We follow the notation and definitions of Bertsekas [71]. However, in CTP, decisions about the structure of the tree source used for encoding the input sequence $x$ influence the coding length in a deterministic manner. Therefore, we describe DP in a deterministic framework.

The basic model in DP is a discrete time dynamic system that evolves according to some state space and is influenced by decisions made at each of the discrete time instances. The system has the form

$$X_{k+1} = f_k(X_k, U_k), \quad k \in \{0, \ldots, N - 1\},$$
where $X_k$ is the state at time $k$, $U_k$ is the decision variable at time $k$, $f_k$ is a next-state function, and $N$ is a time horizon.

There is also an additive cost function associated with the dynamic system. At time $k$, the cost incurred is $g_k(X_k, U_k)$. The total cost is

$$g_N(X_N) + \sum_{k=0}^{N-1} g_k(X_k, U_k),$$

where $g_N(X_N)$ is some terminal cost at the end of the process.

We want to develop an optimal control policy that will minimize the total cost. Denote the control law at time $k$ by $\mu_k$. The control law $\mu_k$ maps the state $X_k$ to a decision $U_k$, i.e.,

$$U_k = \mu_k(X_k).$$

Define the sequence of control laws at all the discrete times in our time horizon as

$$\pi \triangleq \{\mu_0, \ldots, \mu_{N-1}\}.$$  

We call $\pi$, which is a sequence of control laws, a control policy. Given an initial state $X_0$ and policy $\pi$, we will incur an additive cost

$$J_\pi(X_0) = g_N(X_N) + \sum_{k=0}^{N-1} g_k(X_k, \mu_k(X_k)),$$

where $X_k$ evolves according to

$$X_{k+1} = f_k(X_k, \mu_k(X_k)).$$

The optimal cost, which is denoted by $J^*(X_0)$, is defined as

$$J^*(X_0) \triangleq \min_{\pi \in \Pi} J_\pi(X_0),$$

where $\Pi$ is the set of all possible control policies. Accordingly, we define the optimal policy associated with the initial state $X_0$ as

$$\pi^*(X_0) \triangleq \arg \min_{\pi \in \Pi} J_\pi(X_0).$$
Note that, with these definitions, \( J^\ast(X_0) = J^\ast_\ast(X_0) \). The basic theorem of DP states that \( J^\ast(X_0) = J_0(X_0) \), where

\[
J_N(X_N) \triangleq g_N(X_N)
\]

and

\[
J_k(X_k) = \min_{U_k} \{ g_k(X_k, U_k) + J_{k+1}(f_k(X_k, U_k)) \},
\]

where \( k \in \{0, \ldots, N - 1\} \), and we construct \( J_0(X_0) \) by proceeding backwards in time from \( N - 1 \) to 0.

### A.2 Review of Context Tree Pruning (CTP)

We now express the length required for encoding an input \( x \) with a tree source model structure \( S \). In order to keep the presentation simple, we restrict our discussion to a binary alphabet \( \mathcal{X} = \{0, 1\} \); the generalization to nonbinary alphabets is straightforward. Recall that \( S \) is a set of sequences that is complete and proper, so the sequences of \( S \) can be arranged as leaves on a tree. Define the set of internal nodes of the tree source as \( \overline{S} \), a set of finite-length sequences that correspond to internal nodes of \( S \). Note that for \( s \in S \), \( \overline{S} \) contains all the suffixes of \( s \), i.e., \( s_k^{[s]} \in \overline{S} \) for \( k \in \{2, \ldots, |s|\} \). In this appendix, we express the coding length required for encoding \( x \) with the unbounded variant of BWT-MDL; the generalization to the bounded variant is straightforward.

In order to describe \( x \) with \( S \), we need to first describe the structure \( S \) and then describe \( x \) itself. The structure \( S \) is described with the natural code. The natural code describes each node of the tree source structure with one bit. The leaves of the tree source correspond to the sequences of \( S \); the internal nodes of the tree source correspond to the sequences of \( \overline{S} \). Therefore,

\[
|\text{natural}_S| = \sum_{s \in S \cup \overline{S}} 1.
\]

The description of \( x \) itself has two components.
• **Symbols with known context** - we describe all the symbols that were generated by a leaf state \( s \in \mathcal{S} \) using Kruchevsky-Trofimov probability estimates [44]. In Section 2.2.3, we showed that the coding length \( KT_s \) required for encoding all such symbols satisfies

\[
KT_s = -\log \left( \frac{\pi^{-\frac{|\mathcal{X}|}{2}} \Gamma \left( \frac{|\mathcal{X}|}{2} \right) \prod_{a \in \mathcal{X}} \Gamma \left( n_a^\mathcal{S}(x) + \frac{1}{2} \right)}{\Gamma \left( \frac{|\mathcal{X}|}{2} + n_s(x) \right)} \right)
\]

\[
= -\log \left( \frac{\Gamma \left( n_s^\mathcal{S}(x) + \frac{1}{2} \right) \Gamma \left( n_s^\mathcal{S}(x) + \frac{1}{2} \right)}{\pi \Gamma \left( n_s^\mathcal{S}(x) + n_s^\mathcal{S}(x) + 1 \right)} \right).
\]

• **Symbols with unknown context** - if \( x_1^{[s]} = s \in \mathcal{S} \), we encode \( x_{[s]+1} \) directly, using one bit.

We now describe how the coding length can be broken up, such that every node of a tree source is associated with part of the coding length. If \( s \in \mathcal{S} \), i.e., \( s \) is a leaf, we associate \( s \) with a cost \( C_s = 1 + KT_s \). If \( s \in \overline{\mathcal{S}} \), i.e., \( s \) is an internal node, we associate \( s \) with a cost \( C_s = 1 + C_{0s} + C_{1s} + 1_{\{x_1^{[s]} = s\}} \), where \( C_{0s} \) and \( C_{1s} \) are the costs of its children nodes.

Note that \( C_{\lambda} \), the cost associated with the root node, satisfies

\[
C_{\lambda} = \sum_{s \in \mathcal{S}} KT_s + \sum_{s \in \mathcal{S} \cup \overline{\mathcal{S}}} 1 + \sum_{s \in \overline{\mathcal{S}}} 1_{\{x_1^{[s]} = s\}}.
\]

Therefore, \( C_{\lambda} \) is equal to the coding length required to describe \( x \) with \( \mathcal{S} \).

We now point out that, if there are \( n_s(x) \leq 1 \) symbols that were generated by state \( s \), then \( KT_s = n_s(x) \). Keeping children states of \( s \) would only increase the length of the natural code, so \( s \) is a leaf in this case. Therefore, in order to determine the MDL structure \( \mathcal{S}^* \), we can restrict our attention to the class of tree source structures \( \mathcal{S} \) that do not have internal nodes \( s \in \overline{\mathcal{S}} \) such that \( n_s(x) \leq 1 \). Accordingly, once we determine that \( n_s(x) \leq 1 \), we do not consider \( 0s \) and \( 1s \) as potential states.

Based on these considerations, the unbounded variant of Basic BWT-MDL proceeds as follows.
• We first process state \( s = \lambda \).

• If \( n_s(x) \leq 1 \), we only consider \( s \) as a potential leaf, and do not process children states. The coding length required for such a state \( s \) is \( C_s = 1 + n_s(x) \).

• If \( n_s(x) > 1 \), we consider \( s \) as a potential leaf or a potential internal node.
  
  – If \( s \) is a leaf, \( C_s = 1 + KT_s \).
  
  – If \( s \) is an internal node, \( C_s = 1 + C_{0s} + C_{1s} + 1_{\{x_1^{|s} = s\}} \), where \( C_{0s} \) and \( C_{1s} \) are determined by recursively processing the states \( 0s \) and \( 1s \) respectively.

For each such node \( s \), the unbounded variant decides whether \( s \) should be a leaf or internal node by comparing \( 1 + KT_s \) to \( 1 + C_{0s} + C_{1s} + 1_{\{x_1^{|s} = s\}} \). If \( KT_s > C_{0s} + C_{1s} + 1_{\{x_1^{|s} = s\}} \), \( s \) is an internal node, else \( s \) is a leaf.

In Section A.4, we will see that comparing \( KT_s \) to \( C_{0s} + C_{1s} + 1_{\{x_1^{|s} = s\}} \) yields an optimal decision policy.

### A.3 Notation

Recall from Section 1.3.5 that a context tree contains nodes that correspond to sequences that appeared at least twice in \( x \), i.e., the context tree stores information up to branches that correspond to a single symbol of \( x \). Denote the set of sequences that correspond to the nodes of the context tree by \( Y \). Note that \( Y \) is also equal to the set of states that the the unbounded variant of Basic BWT-MDL considered as a potential leaf or internal node.

Define \( Z \) by appending to \( Y \) the children states of every leaf of \( Y \), i.e., \( Z \triangleq \cup_{s \in Y} \{ s, 0s, 1s \} \). By including \( 0s \) and \( 1s \) in \( Z \), we make sure that \( Z \) is the set of all states that are processed by the unbounded variant of Basic BWT-MDL. Note that \( Z \) is a tree because for every \( s \in Z \), if \( s \in Y \) we have \( 0s, 1s \in Z \), else \( s \) is a leaf of \( Z \). Let \( Z(s) \) be the
subtree of $Z$ that is rooted at node $s$. Let $Y(s)$ be the portion of $Y$ that lies in $Z(s)$, i.e., $Y(s) \triangleq Z(s) \cap Y$. Note that, if $s$ is a leaf of $Z$, $Z(s) = \{s\}$ and $Y(s) = \{\}$. If $s$ is an internal node of $Z$, then $s \in Y(s)$.

Define $U(s)$ as a set of decisions for each $s' \in Y(s)$; we denote the decision for $s'$ by $U_{s'}$. The set $U(s)$ determines what tree source substructure $S(s)$ (rooted at $s$) we are considering for the description of symbols that were generated by state $s$. In particular, given $U(s)$, we recursively define $S(s)$ as follows.

- $s \in S(s)$.
- For $s' \in Y(s)$, if $s' \in S(s)$ and $U_{s'} = \text{"keep"}$, then $0s', 1s' \in S(s)$.
- For $s' \in Y(s)$, if $U_{s'} = \text{"prune"}$, then $0s', 1s' \in Z(s)$ but $0s', 1s' \notin S(s)$.

Finally, define $N(s)$ as a set of symbol counts for each node of $Z(s)$, i.e., $N(s) \triangleq \cup_{s' \in Z(s)} \{n^0_{s'}(x), n^1_{s'}(x)\}$. Using the definitions that we have made thus far, $C_s$ is the coding length required to describe the symbols generated by $s$ with $S(s)$. We now define the MDL subtree structure $S^*(s)$ as the subtree structure that yields the shortest description length $C^*_s$ for symbols generated by $s$.

### A.4 Correspondence Between CTP and DP

We will show that the decision policy of the unbounded variant of Basic BWT-MDL is optimal in the sense that for state $s$ it determines $S^*(s)$ and $C^*_s$. We connect several DP concepts to CTP.

- The state of the dynamic system is defined as

$$X_s \triangleq \{Z(s), Y(s), U(s), N(s)\}.$$  

Note that $X_s$ includes all the information on decisions and symbol counts necessary to compute $KT_{s'}$ and $C_{s'}$ for any $s' \in Z_s$. 

187
• The next-state functions involve transitions from state $s$ to either of the children states 0s or 1s. Define

$$f^0_s(X_s) \triangleq X_{0s} \text{ and } f^1_s(X_s) \triangleq X_{1s}.$$  

Note that $X_{0s} = \{Z(0s), Y(0s), U(0s), N(0s)\}$ and $X_{1s} = \{Z(1s), Y(1s), U(1s), N(1s)\}$ can both be determined from $X_s$.

• The control decision for each $s' \in Y(s)$ is $U_{s'} \in \{\text{“keep”, “prune”}\}$. In Section A.3, we explained how $U(s)$ enables us to determine $S(s)$.

• For each $s \in Y(s)$, the additive cost function is defined as

$$g_s(X_s, U_s) \triangleq \begin{cases} 
1 + 1_{\{s\_1 = s\}} & U_s = \text{“keep”} \\
1 + KT_s - J_\pi(X_{0s}) - J_\pi(X_{1s}) & U_s = \text{“prune”} 
\end{cases}, \quad (A.1)$$

where $J_\pi(\cdot)$ is defined below. The terminal cost function for a leaf of $Z(s)$, i.e., $s \in Z(s)$ and $s \not\in Y(s)$, is defined as

$$g_s(X_s) \triangleq 1 + KT_s.$$  

• A control law for state $X_s$ is denoted by $\mu(X_s)$. A control policy for all states is denoted by $\pi$.

• Given an initial state $X_s$ and policy $\pi$, we will incur an additive cost

$$J_\pi(X_s) = \sum_{s' \in Z(s), s' \not\in Y(s)} g_s(X_s') + \sum_{s' \in Y(s)} g_s(X_s', \mu(X_s')).$$  

An important observation is that $J_\pi(X_s)$ can also be defined recursively according to the next-state functions.

$$J_\pi(X_s) = \begin{cases} 
g_s(X_s) & s \in Z(s) \text{ and } s \not\in Y(s) \\
g_s(X_s, U_s) + J_\pi(f^0_s(X_s)) + J_\pi(f^1_s(X_s)) & s \in Y(s) 
\end{cases}.$$
Using (A.1), we have

\[
J_\pi(X_s) = \begin{cases} 
  g_s(X_s) & s \in Z(s) \text{ and } s \not\in Y(s) \\
  1 + J_\pi(X_{0s}) + J_\pi(X_{1s}) + 1_{\{x_1^s|s}\} & s \in Y(s) \text{ and } U_s = \text{"keep"} \\
  1 + KT_s & s \in Y(s) \text{ and } U_s = \text{"prune"} 
\end{cases}
\]

For \( s \in Z(s) \) but \( s \not\in Y(s) \), \( s \) is only considered as a leaf state, and we have \( C_s = 1 + KT_s = g_s(X_s) = J_\pi(X_s) \). Recall also that for \( s \in Y(s) \) and \( U_s = \text{"keep"} \), we had \( C_s = 1 + C_{0s} + C_{1s} + 1_{\{x_1^s|s\}} \); for \( s \in Y(s) \) and \( U_s = \text{"prune"} \), we had \( C_s = 1 + KT_s \). The conclusion is that \( C_s = J_\pi(X_s) \). Furthermore, \( C_s^* = J^*(X_s) \).

With the mathematical framework that we described above, the DP algorithm [71, p.23] can now be formulated as follows. For leaf states of \( Z_s \), i.e., \( s \in Z(s) \) but \( s \not\in Y(s) \), define

\[
\overline{J}(X_s) \triangleq g_s(X_s).
\]

For internal nodes of \( Z_s \), i.e., \( s \in Y(s) \), define

\[
\overline{J}(X_s) \triangleq \min_{U_s \in \{\text{"prune"}, \text{"keep"}\}} \{g_s(X_s, U_s) + \overline{J}(f^0_s(X_s)) + \overline{J}(f^1_s(X_s))\}
\]

\[
= \min_{U_s \in \{\text{"prune"}, \text{"keep"}\}} \{g_s(X_s, U_s) + \overline{J}(X_{0s}) + \overline{J}(X_{1s})\}.
\]

Following Bertsekas [71, p.23],

\[
J^*(X_s) = \overline{J}(X_s).
\]

We now explain why this result indicates that CTP is a form of a DP algorithm. For a leaf node \( s \), i.e., \( s \in Z(s) \) but \( s \not\in Y(s) \), \( C_s^* = g_s(X_s) = 1 + KT_s \). For an internal node \( s \), we have

\[
J^*(X_s) = \min_{U_s \in \{\text{"prune"}, \text{"keep"}\}} \{g_s(X_s, U_s) + J^*(X_{0s}) + J^*(X_{1s})\}
\]

\[
= \min\{1 + J^*(X_{0s}) + J^*(X_{1s}) + 1_{\{x_1^s|s\}}, 1 + KT_s\}
\]

\[
= 1 + \min\{C_{0s}^* + C_{1s}^* + 1_{\{x_1^s|s\}}, KT_s\}. \quad (A.2)
\]
But the condition (A.2) is identical to the decision performed by the unbounded variant of Basic BWT-MDL. We conclude that the decision policy carried out by the unbounded variant of BWT-MDL is identical to the optimal policy $\pi^*$. Therefore, CTP is a form of a DP algorithm for coding length minimization.
APPENDIX B

EXAMPLE OF THE MDL ROUTINE

We now run the bounded depth MDL routine on the sequence $x = x_{11}^1$ from Example 1, using $D_{\text{max}} = 2$. Instead of using coding lengths $KT_s$ and $MDL_s$, we use probabilities $KTP_s = 2^{-KT_s}$ and $MDLP_s = 2^{-MDL_s}$, respectively. The example follows the pseudocode in Figure 2.2.

The computation of $MDL(\lambda, 1, 11)$ begins in Line 9 of Figure 2.2. We have $V(11) = 1$, since $x_1$ appears in $y_{11}$. Hence, $V(b) = 1 = |s| + 1$, and $\kappa \leftarrow 1$ (Line 9). On the other hand, $|s| = |\lambda| = 0 < D_{\text{max}} - 1$, and $\mu \leftarrow 0$. In Line 10, we determine that $m = 2$ because $V(2) = 2$ ($x_2$ appears in $y_2$) and $x_{V(2)-|\lambda|-1} = x_1 = 0$, while $V(3) = 5$ ($x_5$ appears in $y_3$) and $x_{V(3)-|\lambda|-1} = x_4 = 1$. We examine $\text{segment}_0 = y_2^1$ and $\text{segment}_1 = y_3^{10}$ in Lines 12 and 17, respectively.

- $\text{Segment}_0$ - we run $MDL(0,1,2)$. We have $V(2) = 2$ ($x_2$ appears in $y_2$). Hence, $V(b) = 2 = |s| + 1$, and $\kappa \leftarrow 1$ (Line 9). On the other hand, $|s| = |0| = 1 = D_{\text{max}} - 1$, and $\mu \leftarrow 1$. In Line 10, we determine that $m < 1$ because $V(1) = 4$ ($x_4$ appears in $y_1$), and $x_{V(1)-|0|-1} = x_2 = 1$. $\text{Segment}_0$ is empty (Line 13), so $MDLP_{00} = 2^{\mu-1} - 1 = 1$, $\{n_{00}^\alpha(x)\}_{\alpha \in \mathcal{X}} = (0,0)$, natural$_{00} = \lambda$, and $B_{00} = (-1,-1)$ (Line 14). $\text{Segment}_{10} = y_1^1$ is evaluated with $MDL(10,1,1)$ in Line 17. $\text{Segment}_{10} = y_1^1$ satisfies $|10| = D_{\text{max}}$ (Line 2), so its symbol counts are $\{n_{10}^\alpha(x)\}_{\alpha \in \mathcal{X}} = (0,1)$ (Line 3).
and $MDLP_{10} = KTP_{10} = \frac{1}{2}$ (Line 5). We set $natural_{10} = \lambda$ (Line 6) and $B_{10} = (1, 1)$ (Line 7), and return these values on Line 32, passing control back to Line 17.

We have $\{n^0_0(x)\}_{x} = (0, 2)$ (Lines 20-21), and $KTP_0 = \frac{3}{8}$ (Line 22). On the other hand, splitting $segment_0$ corresponds to probability $MDLP_{00} \cdot MDLP_{10} \cdot 2^{-\kappa} = \frac{1}{4}$. Since $\frac{1}{4} < \frac{3}{8}$ (Line 23), $MDLP_0 = \frac{1}{2} \cdot \frac{3}{8} = \frac{3}{16}$ (Line 28), so $segment_{00}$ and $segment_{10}$ are merged into $segment_0$, $natural_0 = 0$ (Line 29), and $B_0 = (1, 2)$ (Line 30).

**Segment$_1$** - we run $MDL(1, 3, 10)$. We have $V(10) = 11$ ($x_{11}$ appears in $y_{10}$). Hence, $V(b) = 11 \neq |s| + 1$, and $\kappa \leftarrow 0$ (Line 9). On the other hand, $|s| = |1| = 1 = D_{max} - 1$, and $\mu \leftarrow 1$. In Line 10, we determine that $m = 4$ because $V(4) = 3$ ($x_3$ appears in $y_1$) and $x_{V(4)-|p|-1} = x_1 = 0$, while $V(5) = 6$ ($x_6$ appears in $y_5$) and $x_{V(5)-|p|-1} = x_4 = 1$. We examine $segment_{01} = y^A_3$ and $segment_{11} = y^B_5$ in Lines 12 and 17, respectively. $Segment_{01} = y^A_3$ is evaluated with $MDL(01, 3, 4)$, and it satisfies $|01| = D_{max}$ (Line 2), so its symbol counts are $\{n^0_{01}(x)\}_{x} = (1, 1)$ (Line 3), $MDLP_{01} = KTP_{01} = \frac{1}{8}$ (Line 5), and we set $natural_{01} = \lambda$ (Line 6) and $B_{01} = (3, 4)$ (Line 7). $Segment_{11} = y^B_5$ is evaluated with $MDL(11, 5, 10)$, and it satisfies $|11| = D_{max}$ (Line 2), so its symbol counts are $\{n^0_{11}(x)\}_{x} = (0, 6)$ (Line 3), $MDLP_{11} = KTP_{11} = \frac{231}{256}$ (Line 5), and we set $natural_{11} = \lambda$ (Line 6) and $B_{11} = (5, 10)$ (Line 7). We have $\{n^0_{1}(x)\}_{x} = (1, 7)$ (Lines 20-21), and $KTP_1 = \frac{429}{256}$ (Line 22). On the other hand, splitting $segment_1$ corresponds to probability $MDLP_{01} \cdot MDLP_{11} \cdot 2^{-\kappa} = \frac{231}{256}$. Since $\frac{231}{256} > \frac{429}{256}$ (Line 23), $MDLP_1 = \frac{1}{2} \cdot \frac{231}{256} = \frac{231}{256}$ (Line 24), so $segment_{01}$ and $segment_{11}$ remain in the MDL subtree for $segment_1$, $natural_1 = 1$ (Line 25), and $B_1 = (3, 4)(5, 10)$ (Line 26).

We have $\{n^0_{\lambda}(x)\}_{x} = (2, 9)$ (Lines 20-21), and $KTP_\lambda = \frac{663}{256}$ (Line 22). On the other hand, splitting $segment_\lambda$ corresponds to probability $MDLP_0 \cdot MDLP_1 \cdot 2^{-\kappa} = \frac{663}{256}$. Since $\frac{663}{256} > \frac{663}{256}$ (Line 23), $MDLP_\lambda = \frac{1}{2} \cdot \frac{663}{256} = \frac{663}{256}$ (Line 24), so $segment_0$, $segment_{01}$, and
segment_{11} remain in the MDL tree source structure, \textit{natural}_{\lambda} = 101 \ (\text{Line 25}), and \ B_{\lambda} = (1,2)(3,4)(5,10) \ (\text{Line 26}). The MDL tree source structure in this case is identical to the structure depicted in Figure 1.1, and the natural code is identical to that of Example 6. Note that \ y_{11}, which was generated by an unknown context, does not belong to any of the ordered pairs of \ B_{\lambda}.
APPENDIX C

PROOF OF LEMMA 5

Let \( f(z) \triangleq \ln(2)D(b_z||r_z) \) where \( z \in \mathbb{R}, \ 1 \leq z \leq K \).

\[
f(z) = \ln(2)D \left( \sin^2 \left( \frac{\pi z}{2K} \right) \parallel \sin^2 \left( \frac{\pi(2z - 1)}{4K} \right) \right) \\
= 2\sin^2 \left( \frac{\pi z}{2K} \right) \ln \left[ \frac{\sin \left( \frac{\pi z}{2K} \right)}{\sin \left( \frac{\pi(2z-1)}{4K} \right)} \right] + 2\cos^2 \left( \frac{\pi z}{2K} \right) \ln \left[ \frac{\cos \left( \frac{\pi z}{2K} \right)}{\cos \left( \frac{\pi(2z-1)}{4K} \right)} \right]. \tag{C.1}
\]

The derivative is

\[
\frac{d}{dz}f(z) = \frac{\pi}{K} \left\{ \sin^2 \left( \frac{\pi z}{2K} \right) \left[ \frac{\cos \left( \frac{\pi z}{2K} \right)}{\sin \left( \frac{\pi z}{2K} \right)} - \frac{\cos \left( \frac{\pi(2z-1)}{4K} \right)}{\sin \left( \frac{\pi(2z-1)}{4K} \right)} \right] \right. \\
+ \cos^2 \left( \frac{\pi z}{2K} \right) \left[ \frac{-\sin \left( \frac{\pi z}{2K} \right)}{\cos \left( \frac{\pi z}{2K} \right)} + \frac{-\sin \left( \frac{\pi(2z-1)}{4K} \right)}{\cos \left( \frac{\pi(2z-1)}{4K} \right)} \right] \\
+ 2\sin \left( \frac{\pi z}{2K} \right) \cos \left( \frac{\pi z}{2K} \right) \ln \left[ \frac{\sin \left( \frac{\pi z}{2K} \right) \cos \left( \frac{\pi(2z-1)}{4K} \right)}{\sin \left( \frac{\pi(2z-1)}{4K} \right) \cos \left( \frac{\pi z}{2K} \right)} \right] \left\}.
\]

We now define \( p \triangleq \sin^2 \left( \frac{\pi z}{2K} \right) \) and \( q \triangleq \sin^2 \left( \frac{\pi(2z-1)}{4K} \right) \). Then

\[
\frac{d}{dz}f(z) = \frac{\pi}{K} \left\{ 2\sqrt{p(1-p)} \ln \left( \sqrt{\frac{p(1-q)}{q(1-p)}} \right) + p \left[ \sqrt{\frac{1-p}{p}} - \sqrt{\frac{1-q}{q}} \right] \\
+ (1-p) \left[ \sqrt{\frac{q}{1-q}} - \sqrt{\frac{p}{1-p}} \right] \right\} \\
= \frac{\pi}{K} \left[ \ln \left( \frac{p(1-q)}{q(1-p)} \right) \sqrt{I(p)} \right] - (p-q)\sqrt{I(q)}, \tag{C.2}
\]
where we used (3.5) in (C.2). But (C.2) is nonpositive according to Lemma 12 (see Appendix D), so \( \frac{d}{dz} D(b_z || r_z) \leq 0 \) where \( z \in \mathbb{R}, 0 \leq z \leq K \). \( \square \)
APPENDIX D

LEmMA 12

Lemma 12 The following inequality holds for any $0 < q < p < 1$.

$$\ln \left( \frac{p(1-q)}{q(1-p)} \right) \leq (p-q)\sqrt{I(p)I(q)},$$

where $I(p)$ and $I(q)$ are the Fisher information for Bernoulli sequences (3.5).

Proof: Define

$$f(p,q) \triangleq \ln \left( \frac{p(1-q)}{q(1-p)} \right) - (p-q)\sqrt{I(p)I(q)}.$$

Substituting for the Fisher information (3.5), we have

$$f(p,q) = \ln \left( \frac{p(1-q)}{q(1-p)} \right) - \frac{p-q}{\sqrt{p(1-p)q(1-q)}}.$$

We take the partial derivative with respect to $p$.

$$\frac{\partial}{\partial p} f(p,q) = \frac{1}{p} + \frac{1}{1-p} - \frac{\sqrt{p(1-p)q(1-q)} - (p-q)^{\frac{3}{2}}[p(1-p)q(1-q)]^{-\frac{1}{2}}q(1-q)(1-2p)}{p(1-p)q(1-q)}$$

$$= \frac{1}{p(1-p)} \left\{ 1 - \frac{1}{2} \left[ \frac{p(1-q)}{q(1-p)} + \frac{q(1-p)}{p(1-q)} \right] \right\}$$

$$\leq 0,$$

and the proof is concluded by noting that $f(q,q) = 0$, so

$$\ln \left( \frac{p(1-q)}{q(1-p)} \right) - (p-q)\sqrt{I(p)I(q)} = \int_{q}^{p} \frac{\partial}{\partial u} f(u,q) \, du \leq 0.$$
\text{APPENDIX E}

\text{PROOF OF THEOREM 7}

\[ D_{J_K} = D(b_1 || r_1) \]
\[ = 2 \sin^2 \left( \frac{\pi}{2K} \right) \log \left[ \frac{\sin \left( \frac{\pi}{2K} \right)}{\sin \left( \frac{\pi}{4K} \right)} \right] + 2 \cos^2 \left( \frac{\pi}{2K} \right) \log \left[ \frac{\cos \left( \frac{\pi}{2K} \right)}{\cos \left( \frac{\pi}{4K} \right)} \right] \quad (E.1) \]
\[ = 2 \sin^2 \left( \frac{\pi}{2K} \right) \log \left[ 2 \cos \left( \frac{\pi}{4K} \right) \right] \\
+ 2 \cos^2 \left( \frac{\pi}{2K} \right) \left\{ \log \left[ \cos \left( \frac{\pi}{2K} \right) \right] - \log \left[ \cos \left( \frac{\pi}{4K} \right) \right] \right\} \]
\[ = 2 \left[ \frac{\pi^2}{4K^2} + O \left( \frac{1}{K^4} \right) \right] \left[ 1 + O \left( \frac{1}{K^2} \right) \right] + 2 \left[ 1 + O \left( \frac{1}{K^2} \right) \right] \]
\[ \left\{ \left[ -\frac{\pi^2}{8K^2 \ln(2)} + O \left( \frac{1}{K^4} \right) \right] - \left[ -\frac{\pi^2}{32K^2 \ln(2)} + O \left( \frac{1}{K^4} \right) \right] \right\} \quad (E.2) \]
\[ = \frac{\pi^2}{K^2} \left( \frac{1}{2} - \frac{3}{16 \ln(2)} \right) + O \left( \frac{1}{K^4} \right), \]

where (E.1) is based on (C.1), and (E.2) uses trigonometric identities, which can be verified with Taylor expansions. \qed
APPENDIX F

PROOF OF LEMMA 6

Using the definition (3.29) of $TD_j(b_i, b_{i+j})$ for the second inequality, we only need to show the first inequality $SD(Q^{i+j}_i) \leq TD_j(b_i, b_{i+j})$. The proof is by induction on $j$. For $j = 1$, owing to the monotonicity of the divergence [1], there is a unique $r^*_i \in [b_i, b_{i+1}]$ such that $D(b_i || r^*_i) = D(b_{i+1} || r^*_i) = TD_1(b_i, b_{i+1})$. If $r^*_i \neq r^*_i$, $SD(Q^{i+1}_i) < TD_1(b_i, b_{i+1}) < WD(Q^{i+1}_i)$. Assuming that the result holds for $j$, we must prove that it also holds for $j + 1$. Consider a quantizer $Q^{i+j+1}_i$. We have

$$TD_{j+1}(b_i, b_{i+j+1}) \geq \min\{TD_j(b_i, b_{i+j}), TD_1(b_{i+j}, b_{i+j+1})\} \quad \text{(F.1)}$$

$$\geq \min\{SD(Q^{i+j}_i), SD(Q^{i+j+1}_i)\} \quad \text{(F.2)}$$

$$= SD(Q^{i+j+1}_i),$$

where (F.1) is obtained by noting that if $b_{i+j} \neq b^*_i$, where $b^*_i$ is the bin edge in $Q^{i+1}_j(b_i, b_{i+j+1})$, then either $TD_j(b_i, b_{i+j})$ or $TD_1(b_{i+j}, b_{i+j+1})$ are smaller than $TD_{j+1}(b_i, b_{i+j+1})$. The second step (F.2) holds because the inductive step implies $SD(Q^{i+j}_i) \leq TD_j(b_i, b_{i+j})$ and the basis of the induction implies $SD(Q^{i+j+1}_i) \leq TD_1(b_{i+j}, b_{i+j+1})$.

$\square$

198
APPENDIX G

DERIVATION OF (3.35) AND (3.39)

Following (3.30), define \( \tilde{s} \) as

\[
\tilde{s} = \frac{2}{\pi} \sqrt{K}.
\]  

(G.1)

We first compute \( D(b_{s}|r_{s+1}) \) ((3.22) and (3.23) do not require \( k \in \mathbb{N} \)), and then relate the result to \( D(b_{s}|r_{s+1}) \).

\[
D(b_{s}|r_{s+1}) = D \left( \sin^2 \left( \frac{\pi \tilde{s}}{2K} \right) \left| \sin^2 \left( \frac{\pi (2\tilde{s} + 1)}{4K} \right) \right. \right)
\]

\[
= D \left( \sin^2 \left( \frac{1}{\sqrt{K}} \right) \left| \sin^2 \left( \frac{1}{\sqrt{K}} + \frac{\pi}{4K} \right) \right. \right)
\]

\[
= 2 \sin^2 \left( \frac{1}{\sqrt{K}} \right) \log \left[ \frac{\sin \left( \frac{1}{\sqrt{K}} \right)}{\sin \left( \frac{1}{\sqrt{K}} + \frac{\pi}{4K} \right)} \right] + 2 \cos^2 \left( \frac{1}{\sqrt{K}} \right) \log \left[ \frac{\cos \left( \frac{1}{\sqrt{K}} \right)}{\cos \left( \frac{1}{\sqrt{K}} + \frac{\pi}{4K} \right)} \right].
\]  

(G.2)

We now use trigonometric and logarithmic identities that can be verified with Taylor expansions.

\[
\sin^2 \left( \frac{1}{\sqrt{K}} \right) = \frac{1}{K} + O \left( \frac{1}{K^2} \right).
\]  

(G.3)

\[
\log \left[ \frac{\sin \left( \frac{1}{\sqrt{K}} \right)}{\sin \left( \frac{1}{\sqrt{K}} + \frac{\pi}{4K} \right)} \right] = \frac{1}{\ln(2)} \left\{ -\frac{\pi}{4\sqrt{K}} + \frac{\pi^2}{32K} + O \left( \frac{1}{K^{1.5}} \right) \right\}.
\]  

(G.4)

\[
\cos^2 \left( \frac{1}{\sqrt{K}} \right) = 1 + O \left( \frac{1}{K} \right).
\]  

(G.5)
\[
\log \left[ \frac{\cos \left( \frac{1}{\sqrt{K}} \right)}{\cos \left( \frac{1}{\sqrt{K}} + \frac{\pi}{4K} \right)} \right] = \frac{1}{\ln(2)} \left\{ \frac{\pi}{4K^{1.5}} + \frac{\pi^2}{32K^2} + O \left( \frac{1}{K^{2.5}} \right) \right\}. 
\]

Substituting (G.3), (G.4), (G.5), and (G.6) into (G.2), we have

\[
D(b_{\bar{s}}\|r_{\bar{s}+1}) = \frac{\pi^2}{8K^2\ln(2)} + O \left( \frac{1}{K^{2.5}} \right).
\]

Using (3.30) and (G.1), \( s \geq \bar{s} \). But \( D(b_k\|r_{k+1}) \) is monotone increasing in \( k \) (Lemma 5), so \( D(b_{\bar{s}}\|r_{\bar{s}+1}) \geq D(b_{\bar{s}}\|r_{\bar{s}+1}) \). Therefore we have (3.35).

The derivation of \( D(b_{\bar{s}}\|r_{\bar{s}}) \) is similar to that of \( D(b_{\bar{s}}\|r_{\bar{s}+1}) \). By showing that

\[
D(b_{\bar{s}}\|r_{\bar{s}}) = \frac{\pi^2}{8K^2\ln(2)} + O \left( \frac{1}{K^{2.5}} \right),
\]

and noting that \( D(b_k\|r_k) \) is monotone decreasing in \( k \), we have (3.39).
APPENDIX H

PROOF OF THEOREM 11

We first use the construction $\tilde{J}_K$ from the proof of Theorem 10 to provide a QP code whose WCR is upper bounded by $1.047 + o(1)$ bits above Rissanen’s bound. We complete the proof by extending the lower bound of Theorem 10 to the WCR.

Construction of near-optimum code: Using Lemma 3, the WCR of the construction $\tilde{J}_K$ from the proof of Theorem 10 is upper bounded by $1.047 + o(1)$ bits above Rissanen’s bound.

Lower bound on WCR: Lemma 6 can be extended to operational divergences (WCBDO’s) using a similar proof. Using the operational equivalent of (3.31), we can lower bound the SDO of $\tilde{J}_{K-2s}$ over the interval $\hat{\Gamma}$ by $D(b_s + \frac{1}{N}\|r_{s+1})$. Following the derivation of (3.34), we thus have

$R^O_{\tilde{J}_{K-2s}}(N) \geq \log(K - 2s) + N \cdot D\left(b_s + \frac{1}{N}\|r_{s+1}\right)$.

Using (3.30), $s = O(K^{0.5})$. It follows from (3.22) and (3.23) that $b_s = O(K^{-1})$ and $r_{s+1} = O(K^{-1})$. Using (G.1)-(G.3),

$r_{s+1} - b_s = \sin^2\left(\frac{1}{\sqrt{K}} + \frac{\pi}{4K}\right) - \sin^2\left(\frac{1}{\sqrt{K}}\right)
= \frac{\pi}{2K^{1.5}} + O\left(\frac{1}{K^2}\right)$. 

201
Using $|s - \bar{s}| < 1$, we have $r_{s+1} - b_s = O(K^{-1.5})$. Because

$$\frac{\partial}{\partial p} D(p\|q) = \log \left( \frac{p(1 - q)}{q(1 - p)} \right),$$

we have

$$D \left( b_s + \frac{1}{N} \parallel r_{s+1} \right) = D(b_s\|r_{s+1}) + \int_{b_s}^{b_s + \frac{1}{N}} \log \left( \frac{p(1 - r_{s+1})}{r_{s+1}(1 - p)} \right) \, dp. \quad (H.1)$$

But for $p \in (b_s, r_{s+1})$ we have

$$\frac{p(1 - r_{s+1})}{r_{s+1}(1 - p)} - 1 = \frac{r_{s+1} + O(K^{-1.5})}{1 + O(K^{-1})} - 1$$
$$= \left[ 1 + \frac{O(K^{-1.5})}{O(K^{-1})} \right] - 1$$
$$= O(K^{-0.5}),$$

so

$$\int_{b_s}^{b_s + \frac{1}{N}} \log \left( \frac{p(1 - r_{s+1})}{r_{s+1}(1 - p)} \right) \, dp = O(N^{-1}K^{-0.5}). \quad (H.2)$$

Combining (3.35) with (H.1) and (H.2),

$$D \left( b_s + \frac{1}{N} \parallel r_{s+1} \right) \geq \frac{\pi^2}{8K^2 \ln(2)} + O \left( \frac{1}{K^{2.5}} \right) + O(N^{-1}K^{-0.5}).$$

The remainder of the proof is similar to that of Theorem 10. \hfill \Box
APPENDIX I

PROOF OF THEOREM 12

Following (3.6), let \( \tilde{R}^{|\mathcal{X}|} \) be the integral over the \((|\mathcal{X}| - 1)\)-dimensional probability simplex, i.e.,

\[
\tilde{R}^{|\mathcal{X}|} = \int_{\Gamma} \frac{\sqrt{I(\Theta)}}{d\theta} d\theta \\
= \int_{\theta^0=0}^{\frac{1}{2}} \int_{u^1=0}^{1-\theta^0} \cdots \int_{u^{|\mathcal{X}|-2}=0}^{1-\theta^{|\mathcal{X}|-2}} \frac{d\theta^0 \cdots d\theta^{|\mathcal{X}|-2}}{\sqrt{\prod_{i=0}^{|\mathcal{X}|-1} \theta^i}}. 
\]

We express \( \tilde{R}^{|\mathcal{X}|} \) in terms of \( \tilde{R}^{|\mathcal{X}|-1} \) by substituting the variables \( u^i \triangleq \frac{\theta^i}{1-\theta^0} \) for \( i \in \{1, \ldots, |\mathcal{X}|-1\} \).

\[
\tilde{R}^{|\mathcal{X}|} = \int_{\theta^0=0}^{\frac{1}{2}} \int_{u^1=0}^{1} \cdots \int_{u^{|\mathcal{X}|-2}=0}^{1-\theta^{|\mathcal{X}|-2}} \frac{d\theta^0 \cdots d\theta^{|\mathcal{X}|-2}}{\sqrt{\prod_{i=1}^{|\mathcal{X}|-1} (1-\theta^0) u^i}} \\
= \tilde{R}^{|\mathcal{X}|-1} \int_{\theta^0=0}^{\frac{1}{2}} \frac{(1-\theta^0)^{|\mathcal{X}|-2}}{\sqrt{\theta^0}} d\theta^0. \tag{1.2}
\]

Using the substitution \( \theta^0 = \sin^2(\phi) \), (1.2) reduces to

\[
\tilde{R}^{|\mathcal{X}|} = 2 \tilde{R}^{|\mathcal{X}|-1} \int_{\phi=0}^{\frac{\pi}{2}} \cos(\phi)^{|\mathcal{X}|-2} d\phi. \tag{1.3}
\]

Next, using the standard result

\[
\int_{\phi=0}^{\frac{\pi}{2}} \cos^k(\phi) d\phi = \frac{\sqrt{\pi} \Gamma \left( \frac{k}{2} + 1 \right)}{2 \Gamma \left( \frac{k}{2} + 1 \right)}
\]

203
we obtain
\[ \tilde{R}^{\frac{1}{2}} = \sqrt{\frac{\Gamma\left(\frac{|x|}{2} - \frac{1}{2}\right)}{\Gamma\left(\frac{|x|}{2}\right)}} \tilde{R}^{\frac{|x|}{2}} \]\n
and the basis for the recursion is \( \tilde{R}^2 = \pi \). Therefore,

\[ \tilde{R}^{\frac{|x|}{2}} = \left[ \sqrt{\frac{\Gamma\left(\frac{|x|}{2} - \frac{1}{2}\right)}{\Gamma\left(\frac{|x|}{2}\right)}} \right] \left[ \sqrt{\frac{\Gamma\left(\frac{|x|}{2} - 1\right)}{\Gamma\left(\frac{|x|}{2} - \frac{3}{2}\right)}} \right] \cdots \left[ \sqrt{\frac{\Gamma(1)}{\Gamma(1.5)}} \right] \pi \]

\[ = \frac{\pi^{\frac{|x|}{2}}}{\Gamma\left(\frac{|x|}{2}\right)}. \]

(I.4)

The bound (3.40) is obtained by combining (3.6), (I.1), and (I.4).
APPENDIX J

PROOF OF LEMMA 7

Using (3.44),

\[ l_n(x) = - \sum_{j=0}^{|\mathcal{X}|-1} n^j(x) \log(\hat{\theta}^j) \]

\[ = - \sum_{j=0}^{|\mathcal{X}|-1} n^j(x) \left[ \log(\hat{\theta}^j) + \sum_{k=0}^{j-1} \log(1 - \hat{\theta}^k) \right]. \]

Noting that for any sequence \( \{g(j, k)\} \)

\[ \sum_{j=0}^{|\mathcal{X}|-1} \sum_{k=0}^{j-1} g(j, k) = \sum_{j=0}^{|\mathcal{X}|-1} \sum_{k=j+1}^{|\mathcal{X}|-1} g(k, j), \]

it follows that

\[ l_n(x) = - \sum_{j=0}^{|\mathcal{X}|-1} \left[ n^j(x) \log(\hat{\theta}^j) + \log(1 - \hat{\theta}^j) \sum_{k=j+1}^{|\mathcal{X}|-1} n^k(x) \right] \]

\[ = - \sum_{j=0}^{|\mathcal{X}|-1} \left[ n^j(x) \log(\hat{\theta}^j) + (N^j(x) - n^j(x)) \log(1 - \hat{\theta}^j) \right] \]

\[ = - \sum_{j=0}^{|\mathcal{X}|-2} \left[ n^j(x) \log(\hat{\theta}^j) + (N^j(x) - n^j(x)) \log(1 - \hat{\theta}^j) \right], \quad (J.1) \]

where the last step follows because \( \hat{\theta}^{|\mathcal{X}|-1} = 1 \) and \( N^{|\mathcal{X}|-1}(x) = n^{|\mathcal{X}|-1}(x) \). Using (3.43), we also have

\[ l_{ML}(x) = - \sum_{j=0}^{|\mathcal{X}|-2} \left[ n^j(x) \log(\hat{v}^j) + (N^j(x) - n^j(x)) \log(1 - \hat{v}^j) \right]. \]

205
The redundancy of encoding with normalized parameters is thus

\[ R_n(x) = \sum_{j=0}^{|\mathcal{X}|-2} \left[ n^j(x) \log \left( \frac{v^j}{\hat{v}^j} \right) + (N^j(x) - n^j(x)) \log \left( \frac{1 - v^j}{1 - \hat{v}^j} \right) \right]. \]

The result (3.45) is obtained by using (3.9) and noting that \( n^j(x) = v^j N^j(x) \). \qed
APPENDIX K

EXAMPLE OF THE PMDL ROUTINE

Consider the tree source structure of Figure 1.1 over $\mathcal{X} = \{0, 1\}$. For $N = 16$, the input sequence might be $x_1^{16} = 0101010011111111$. We partition $x$ into $B = 2$ blocks, hence $x(1) = x_1^8 = 01010100$ and $x(2) = x_9^{16} = 11111111$. We use $D_{\text{max}} = 2$, which implies $K_s = 8$.

The block symbol counts for all depth-$D_{\text{max}}$ contexts can be computed by following the pseudocode of Figure 4.4; we omit the details for brevity. The block symbol counts of $\text{PU } 1$ are $(n_{00}^0(1), n_{00}^1(1)) = (0, 0)$, $(n_{01}^0(1), n_{01}^1(1)) = (3, 0)$, $(n_{10}^0(1), n_{10}^1(1)) = (1, 2)$, and $(n_{11}^0(1), n_{11}^1(1)) = (0, 0)$. The block symbol counts of $\text{PU } 2$ are $(n_{00}^0(2), n_{00}^1(2)) = (0, 0)$, $(n_{01}^0(2), n_{01}^1(2)) = (0, 0)$, $(n_{10}^0(2), n_{10}^1(2)) = (0, 0)$, and $(n_{11}^0(2), n_{11}^1(2)) = (0, 6)$.

We now run the PMDL routine, following the pseudocode in Figure 4.5. The computation of $\text{PMDL}(\lambda)$ begins in Line 12. We process $s = 0$ and $s = 1$ in Lines 12 and 17, respectively.

- The computation of $\text{PMDL}(0)$ begins in Line 12. We process $s = 00$ and $s = 10$ in Lines 12 and 17, respectively.

- The computation of $\text{PMDL}(00)$ begins in Line 2. We get $(n_{00}^0(1), n_{00}^1(1)) = (0, 0)$ and $(n_{00}^0(2), n_{00}^1(2)) = (0, 0)$ in parallel from the PUs in Line 4. We compute $(n_{00}^0, n_{00}^1) = (0, 0)$ using (4.2) in Line 6. We compute $l_{00} = 3 =$
\[ \log(K_s) \text{ using (4.11) in Line 7. We set } MDL_{00} = 3 \text{ (Line 8) and } \text{natural}_{00} = \lambda \text{ (Line 9). We return these values in Line 32.} \]

- The computation of \( PMDL(10) \) begins in Line 2. We get \((n_{10}^0(1), n_{10}^1(1)) = (1, 2) \) and \((n_{10}^0(2), n_{10}^1(2)) = (0, 0) \) in parallel from the PUs in Line 4. We compute \((n_{10}^0, n_{10}^1) = (1, 2) \) using (4.2) in Line 6. We compute \( l_{10} = 5.80 \) using (4.11) in Line 7. Note that \( \log(K_s) + n_{10} \hat{H}(\theta_{10}) = 5.76 \), where \( \hat{H}(\theta) = -\theta \log(\theta) - (1-\theta) \log(1-\theta) \). We set \( MDL_{10} = 5.80 \) (Line 8) and \( \text{natural}_{10} = \lambda \) (Line 9). We return these values in Line 32, passing control back to Line 13.

We compute \((n_0^0, n_0^1) = (1, 2) \) using (4.8) in Line 13. We compute \( l_0 = 5.80 \) using (4.11) in Line 14. Because \( l_0 < MDL_{00} + MDL_{10} = 8.80 \), in Line 23 we decide to prune states 00 and 10 into state 0. We set \( MDL_0 = 6.80 \) (Line 19) and \( \text{natural}_0 = 0 \) (Line 20). We return these values in Line 32, passing control back to Line 17.

- The computation of \( PMDL(1) \) begins in Line 12. We process \( s = 01 \) and \( s = 11 \) in Lines 12 and 17, respectively.

- The computation of \( PMDL(01) \) begins in Line 2. We get \((n_{01}^0(1), n_{01}^1(1)) = (3, 0) \) and \((n_{01}^0(2), n_{01}^1(2)) = (0, 0) \) in parallel from the PUs in Line 4. We compute \((n_{01}^0, n_{01}^1) = (3, 0) \) using (4.2) in Line 6. We compute \( l_{01} = 3.04 \) using (4.11) in Line 7. Note that \( l_{01} > \log(K_s) \) because \( n_{01} = 3 > 0 \). We set \( MDL_{01} = 3.04 \) (Line 8) and \( \text{natural}_{01} = \lambda \) (Line 9). We return these values in Line 32.

- The computation of \( PMDL(11) \) begins in Line 2. We get \((n_{11}^0(1), n_{11}^1(1)) = (0, 0) \) and \((n_{11}^0(2), n_{11}^1(2)) = (0, 6) \) in parallel from the PUs in Line 4. We compute \((n_{11}^0, n_{11}^1) = (0, 6) \) using (4.2) in Line 6. We compute \( l_{11} = 3.08 \) using
(4.11) in Line 7. We set $MDL_{11} = 3.08$ (Line 8) and $natural_{11} = \lambda$ (Line 9).

We return these values in Line 32.

We compute $(n_{11}^0, n_{11}^1) = (3, 6)$ using (4.8) in Line 13. We compute $l_{1} = 11.40$
using (4.11) in Line 14. Note that $\log(K_{\delta}) + n_{1} \hat{H}(\theta_{1}) = 11.27$. Because $l_{1} >$
$MDL_{01} + MDL_{11} = 6.12$, in Line 23 we decide to retain states 01 and 11. We set
$MDL_{1} = 7.12$ (Line 16) and $natural_{1} = 1$ (Line 17). We return these values in
Line 32, passing control back to Line 13.

We compute $(n_{\lambda}^0, n_{\lambda}^1) = (4, 8)$ using (4.8) in Line 13. We compute $l_{\lambda} = 14.20$
using (4.11) in Line 14. Note that $\log(K_{\delta}) + n_{\lambda} \hat{H}(\theta_{\lambda}) = 14.02$. Because $l_{\lambda} > MDL_{0} + MDL_{1} = 13.92,$
in Line 23 we decide to retain states 0, 01, and 11. We set $MDL_{\lambda} = 14.92$ (Line 16)
and $natural_{\lambda} = 101$ (Line 17). We return these values in Line 32. The MDL tree source
structure in this case is identical to the tree source structure depicted in Figure 1.1.

Finally, in view of (4.7), encoding $x$ requires $MDL_{\lambda} + B \cdot D_{max} = 18.92$ bits, without
taking the arithmetic coding redundancy into account.
APPENDIX L

PROOF OF LEMMA 8

We will prove

\[ f(k, l + 1) \geq \max_{\substack{p, q \in \mathbb{R}, \ 0 \leq q \leq l, \ 1 \leq p \leq k - 1}} \eta(k, l, p, q). \]

Owing to symmetry of \( p \) and \( k \) in (5.12), we assume \( p \leq \frac{k}{2} \); thus, \( \min\{p, k - p\} = p \), and

\[
\eta(k, l, p, q) = f(p + 1, q) + f(k - p + 1, l - q) + 2p \\
= (p - 1) \log(q + 1) + (k - p - 1) \log(l - q + 1) + \frac{3}{2} l + 2p - 1. \quad \text{(L.1)}
\]

Combining (5.11) and (L.1), it suffices to prove

\[
f(k, l + 1) = (k - 2) \log(l + 2) + \frac{3}{2} l + 1 \\
\geq (p - 1) \log(q + 1) + (k - p - 1) \log(l - q + 1) + \frac{3}{2} l + 2p - 1 \quad \text{(L.2)}
\]

for \( \{p, q : p, q \in \mathbb{R}, \ 0 \leq q \leq l, \ 1 \leq p \leq \frac{k}{2}\} \). The key observation is that (L.1) is linear in \( p \); hence it suffices to prove the inequality for \( p \in \{1, \frac{k}{2}\} \). For \( p = 1 \),

\[
\eta(k, l, p = 1, q) = (k - 2) \log(l - q + 1) + \frac{3}{2} l + 1. \quad \text{(L.3)}
\]

Since the partial derivative with respect to \( q \) is

\[
\frac{\partial}{\partial q} \eta(k, l, p = 1, q) = -\frac{k - 2}{(\ln(2))(l - q + 1)} < 0,
\]

210
\[ \eta(k, l, p = 1, q) \text{ is maximum at } q = 0, \text{ so by (L.3)} \]

\[ \eta(k, l, p = 1, q) \leq (k - 2) \log(l + 1) + \frac{3}{2}l + 1, \]

which satisfies (L.2). For \( p = \frac{k}{2} \),

\[ \eta \left( k, l, p = \frac{k}{2}, q \right) = \left( \frac{k}{2} - 1 \right) \{ \log(q + 1) + \log(l - q + 1) \} + \frac{3}{2}l + k - 1. \] (L.4)

Since \((q + 1) + (l - q + 1) = l + 2\), (L.4) is maximum for \( q = \frac{l}{2} \); thus,

\[ \eta \left( k, l, p = \frac{k}{2}, q \right) \leq 2 \left( \frac{k}{2} - 1 \right) \log \left( \frac{l}{2} + 1 \right) + \frac{3}{2}l + k - 1 \]

\[ = (k - 2) \log(l + 2) - 1 + \frac{3}{2}l + k - 1 \]

\[ = (k - 2) \log(l + 2) + \frac{3}{2}l + 1, \]

which satisfies (L.2). This concludes the proof of Lemma 8.
REFERENCES


213


VITA

Dror Baron was born in Haifa, Israel, in 1973. He received the B.Sc. (summa cum laude) and M.Sc. degrees from the Technion - Israel Institute of Technology, in 1997 and 1999, both in electrical engineering. From 1992 to 1994, he developed software for the Israel Defense Forces. In 1997 he was a teaching assistant in the Electrical Engineering Department at the Technion. From 1997 to 1999 he worked at Witcom Ltd. in modem design. Since 1999 he has been a research assistant at the University of Illinois at Urbana-Champaign, where he is pursuing his Ph.D. His research interests include information theory, data compression, communications systems, signal processing, and hardware design. Dror was a recipient of the 2002 M. E. Van Valkenburg Graduate Research Award and received honorable mention in the Robert Bohrer Memorial Student Workshop in April 2002, both at the University of Illinois. He also participated in the Program for Outstanding Students, comprising approximately 45 undergraduates (out of a student body of 9000), in 1994-1997 at the Technion - Israel Institute of Technology.