Browsing and Searching Compressed Documents

Raymond Wan

Department of Computer Science and Software Engineering
The University of Melbourne

Submitted in partial fulfilment of the requirements of the degree of
Doctor of Philosophy (with coursework component)

December 2003
Abstract

Compression and information retrieval are two areas of document management that exist separately due to the conflicting methods of achieving their goals. This research examines a mechanism which provides lossless compression and phrase-based browsing and searching of large document collections. The framework for the investigation is an existing off-line dictionary-based compression algorithm.

An analysis of the algorithm, supported by previous work and experiments, highlights two factors that are important for retrieval: efficient decoding, and a separate dictionary stream. However, three areas of improvement are necessary, prior to the inclusion of the algorithm into a browsing system.

First, in order to accommodate retrieval, the algorithm must produce a dictionary built up on words, rather than characters. A pre-processing stage is introduced which separates the message into words and non-words, along with word modifiers.

Second, the memory requirements of the algorithm prevent the processing of large documents. Earlier work has proposed a solution which separates the message into individual blocks prior to compression. Here, a post-processing stage is proposed which combines the blocks in a series of phases. Experiments show the trade-offs between the number of phases performed and the improvements in compression levels.

Organisations which provide access to an information retrieval system to users are sometimes concerned with the amount of disk space available. But users have a different point of view and may place response time at a higher priority. Indeed, faster computers and network connections translate into plummeting patience levels of users. The last improvement to the compression algorithm is two new coding schemes which replace the entropy coder that was used in previous work. While deploying them sacrifices compression effectiveness, these two mechanisms offer improved efficiency, as shown through experiments.

With the enhancements to the compression algorithm in hand, a technique to efficiently support phrase browsing is presented. Phrase contexts can be searched and progressively refined through the word modifiers. Because of the three changes to the algorithm, phrases are more visually appealing, larger documents can be processed, and response times are improved.
Declaration

This is to certify that

1. the thesis comprises only my original work towards the PhD except where indicates in the Preface,

2. due acknowledgement has been made in the text to all other material used, and

3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

Raymond Wan
BSc (University of British Columbia, 1997)
Preface

*Publications arising from this thesis.*

The general concept of phrase browsing documents compressed with character-based RE-PAIR was introduced as a poster presentation at the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval [Wan and Moffat, 2001b], and then subsequently expanded for the 2001 Symposium on String Processing and Information Retrieval [Moffat and Wan, 2001].

Merging compressed blocks, as covered in Chapter 5, was presented at the 2002 Symposium for Combinatorial Pattern Matching [Wan and Moffat, 2002].

The results from the experiments of Chapter 8 on compression with HTML hyperlinks was presented at the 2001 Australasian Database Conference [Wan and Moffat, 2001a].
Acknowledgements

As expected, completing work of this magnitude is due to the support of many people. As I didn’t know anyone in Australia before I arrived here for study, there are a lot of relationships I am thankful for.

The continued support over the years from my supervisor, Alistair Moffat, has made much of this thesis possible. I began this degree in the hope of learning, and fortunately, Alistair has made sure of that from the first day until the last.

The past and present members of Alistair’s group have contributed to an excellent support network, and it has been a pleasure to be a part of it. Tim Bell and Owen de Kretser helped me from the very beginning as I struggled with the transition back into university life. Tim’s continued maintenance of the “vi” machines, and calmness whenever I brought them to a grinding halt, is also very much appreciated. Encouragement from Anh Ngoc Vo over the years has certainly kept me going from day-to-day. Mike Ciavarella, Mike Liddell, Yugo Isal, Lily Sun, and Tony Wirth have further contributed to my personal life and my studies.

I’m also fortunate to have worked with (or next to) many other people in the department. Hongjian Fan, Kevin Glynn, Bernard Pope, Robert Shelton, Qun Sun, Xiuzhen (Jenny) Zhang, are only some of the people who have shared an office with me at some stage – four offices in all, spread across two buildings! Inga Sitzmann and Lei Zheng also helped complete my network of postgraduates within the department, while working with Linda Stern on several occasions as a tutor has always been both enjoyable and enriching.

Glen Gibb and Leslie Young have been great friends during my stay in Melbourne, and through them, I have met many others. The love and support given to me from Ju Hyung (Juliet) Lee has only grown since we first met. During that time, she’s been in the unfortunate position of bearing the brunt of the stress, and I thank her for that.
I’m also thankful to many other friends who live overseas, and which I’ve had little opportunity to meet while living here. Wilson Lee has always been a close friend, and his advice is something which I’ve depended on over the years. Joyce Cunningham and Yoshiko Yoshimura provided the motivation to consider further study, while Ed Knorr has always been a mentor to me. Ed’s suggestion to do a research degree instead of a coursework one because “I won’t regret it” was what started everything here. Now that I’m at the end, I can say that he was right after all.

And last, but most of all, to my parents, for their support and caring. Their patience while their son studied on the other side of the world is much appreciated, as is everything else they’ve done for me.
Contents

Contents v

List of Figures xi

List of Tables xv

List of Algorithms xix

1 Introduction 1
  1.1 Information Retrieval ........................................... 4
  1.2 Text Compression .................................................. 7
  1.3 Browsing and Compression ....................................... 8
  1.4 Structure .......................................................... 9

2 Text Compression 13
  2.1 Fundamentals in Compression .................................... 14
  2.2 Compression Systems ............................................. 15
  2.3 Modelling .......................................................... 18
  2.4 Coders ............................................................. 30
  2.5 Compression in Practice ......................................... 36
  2.6 Experimental Framework ....................................... 38
  2.7 Effect of Compression on Browsing ............................ 41

3 Compression by Recursive Pairing 45
3.1 The RE-PAIR Model ........................................... 46
3.2 Analysis of RE-PAIR ........................................ 49
3.3 Coding for RE-PAIR .......................................... 58
3.4 Experiments .................................................... 62
3.5 Related Work .................................................. 65
3.6 Retrieval with RE-PAIR ...................................... 68

4 Selecting Phrases for Browsing .................................. 73
  4.1 Forming Words from Characters ............................ 74
  4.2 Word-aligned RE-PAIR ...................................... 76
  4.3 Word-based RE-PAIR ......................................... 80
  4.4 Punctuation-aligned RE-PAIR ............................... 91
  4.5 Experiments .................................................... 96
  4.6 Phrases for Browsing ......................................... 105

5 Block Merging for Re-Pair ....................................... 109
  5.1 Creating Blocks ............................................... 110
  5.2 Merging Phrase Hierarchies ................................. 114
  5.3 Identifying Old Symbol Pairs .............................. 119
  5.4 Identifying New Symbol Pairs ............................. 136
  5.5 Changes for Punctuation-Aligned RE-PAIR ............... 140
  5.6 Experiments .................................................... 143
  5.7 Appending to a Compressed Document .................... 150
  5.8 Summary ....................................................... 155

6 Coding Considerations .......................................... 159
  6.1 Coding for Compression and Phrase Browsing ............ 160
  6.2 Huffman Coding in Blocks .................................. 164
  6.3 Byte-Aligned Coding in Blocks ............................ 166
  6.4 Experiments .................................................... 172
  6.5 Selecting Coders ............................................. 185
7 Phrase Browsing 189
  7.1 Browsing and Searching Steps .............................. 190
  7.2 Phrase Hierarchy as a Directed Graph ...................... 192
  7.3 Phrase Browsing Example .................................. 196
  7.4 Evaluating RE-PHINE .................................... 201
  7.5 Tools for Retrieval ...................................... 217

8 Compression of HTML Documents 221
  8.1 Related Work ............................................. 222
  8.2 Data Overview ........................................... 223
  8.3 Experimental Framework .................................. 225
  8.4 Experimental Results .................................... 228
  8.5 Summary ................................................... 231

9 Summary 233
  9.1 Putting it all together .................................... 235
  9.2 Towards the Future ....................................... 236

A Notation 239

B Porter Stemming Algorithm 241

Bibliography 247
# List of Figures

1.1 The options available to a library patron ........................................ 3  
1.2 Accessing a repository through an IR system ................................. 5  
1.3 Changes to the ampersand through time ..................................... 8  

2.1 Separation of a compression system ............................................. 17  
2.2 The “Woodchuck” message ....................................................... 18  
2.3 Block sorting example ............................................................ 28  
2.4 Huffman code construction ....................................................... 33  
2.5 Huffman codes as a binary tree ............................................... 34  
2.6 Arithmetic coding example ...................................................... 35  
2.7 Excerpts from the NEWS data set ............................................ 40  

3.1 Main RE-PAIR data structures ................................................... 50  
3.2 Re-using RE-PAIR sequence nodes ........................................... 52  
3.3 Number of active phrases during RE-PAIR execution (WSJ20) .......... 54  
3.4 Chiastic slide example ............................................................ 60  
3.5 Expanding the beginning of the sequence (WSJ20) .......................... 63  

4.1 Expanding the phrases of Table 3.1 ............................................ 77  
4.2 Word-aligned RE-PAIR phrases ................................................ 80  
4.3 The pre-processing stage used by word-based RE-PAIR ................. 82  
4.4 Comparing phrases from the four versions of RE-PAIR (bible.txt) .... 95  
4.5 Compression ratios for the four variants of RE-PAIR (WSJ20) ............ 103
4.6 Expanding the beginning of the sequence with word-aligned and word-based
RE-PAIR (WSJ20) ........................................... 105

5.1 Compression ratios from block-wise character-based RE-PAIR (WSJ508) ... 113
5.2 Combinations of RE-MERGE phases ........................................ 114
5.3 RE-MERGE phase 1 example ............................................... 117
5.4 Symbol dependencies for $P^*$ ........................................... 118
5.5 Representing part of a message as a graph .............................. 124
5.6 Decoding $S_1$ into vertices ............................................... 129
5.7 Adding edges and locating a cut vertex ................................. 132
5.8 Calculating the chance of obtaining a false duplicate .............. 139
5.9 Edge removal for phase 2b and punctuation-aligned RE-PAIR .... 142
5.10 Compression ratios for method E of RE-MERGE (WSJ508) ......... 146
5.11 Compression effectiveness of RE-MERGE for appending text (WSJ508) ... 153
5.12 Compression effectiveness versus throughput for RE-MERGE (NEWS) .... 157

6.1 Indexed Huffman coding ................................................. 165
6.2 Sketch of RE-View blocks ............................................... 168
6.3 Search times for the reduced word sequence for frequency groupings (NEWS) 182
6.4 Search times for the reduced word sequence for symbol length groupings (NEWS) 184

7.1 Sequence of steps performed by RE-PHINE during phrase browsing .... 191
7.2 The six pointers used by a RE-PHINE graph .......................... 193
7.3 The directed graph structure of RE-PHINE, at a larger scale ......... 194
7.4 Sample phrase browsing session of RE-PHINE (NEWS) ............. 198
7.5 Sample decoding session of RE-PHINE (NEWS) ........................ 200
7.6 Definition of a noun phrase .............................................. 202
7.7 Example parsing of a headline with LINK GRAMMAR (WSJ20) ...... 213
7.8 Example article with noun phrases identified (WSJ20) ................ 214

8.1 Percentage of page requests for each browsing session length ....... 227
8.2 First 250 bytes of the two dictionary files for priming . . . . . . . . . . . . . . 228
8.3 Cumulative bandwidths versus browsing session counts . . . . . . . . . . . . . 231

9.1 Sketch of the RE-STORE system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 236

B.1 Porter stemming algorithm – step 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 244
B.2 Porter stemming algorithm – step 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 245
B.3 Porter stemming algorithm – steps 3 and 2 . . . . . . . . . . . . . . . . . . . . . . . . . . 245
B.4 Porter stemming algorithm – step 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 246
B.5 Porter stemming algorithm – step 0 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 246
List of Tables

2.1 Dictionary-based compression using LZ77 .............................. 21
2.2 Dictionary-based compression using LZ78 .............................. 23
2.3 Statistical modelling using PPM ........................................... 27
2.4 Sample static codes ....................................................... 31
2.5 Compression ratios using GZIP, BZIP2, and PPM (all data) .......... 41
2.6 Compression times using GZIP, BZIP2, and PPM (all data) .......... 41

3.1 Sample application of RE-PAIR ........................................... 48
3.2 Trade-offs between decoding time and memory usage for DES-PAIR (WSJ20) . 49
3.3 Memory usage of RE-PAIR data structures ............................. 53
3.4 Statistics from RE-PAIR experiments (LCC, WSJ20) .................. 63
3.5 Compression ratios using RE-PAIR (LCC, WSJ20) .................... 64
3.6 Compression times using RE-PAIR (LCC, WSJ20) .................... 64
3.7 Sample application of SEQUITUR ....................................... 68

4.1 Word-aligned RE-PAIR rules ............................................. 78
4.2 Sample application of word-aligned RE-PAIR .......................... 79
4.3 Unexpected results from the Porter stemming algorithm .............. 85
4.4 Relationship between lexicon size and word token transformations (WSJ20) . 87
4.5 Compression results for word lexicons with respect to themselves (WSJ20) . 90
4.6 Compression results for word lexicons with respect to file size (WSJ20) . 90
4.7 Compression results for the non-word lexicon (WSJ20) ................ 91
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8</td>
<td>Punctuation-aligned RE-PAIR rules</td>
<td>93</td>
</tr>
<tr>
<td>4.9</td>
<td>Statistics for the modifiers and the reduced word sequence (WSJ20)</td>
<td>96</td>
</tr>
<tr>
<td>4.10</td>
<td>Compression ratios for word-based RE-PAIR modifiers (WSJ20)</td>
<td>97</td>
</tr>
<tr>
<td>4.11</td>
<td>Compression times for word-based RE-PAIR modifiers (WSJ20)</td>
<td>98</td>
</tr>
<tr>
<td>4.12</td>
<td>Statistics for the four variants of RE-PAIR (WSJ20)</td>
<td>99</td>
</tr>
<tr>
<td>4.13</td>
<td>Compression ratios of the four versions of RE-PAIR (WSJ20)</td>
<td>100</td>
</tr>
<tr>
<td>4.14</td>
<td>Compression times for character-based and word-aligned RE-PAIR (WSJ20)</td>
<td>104</td>
</tr>
<tr>
<td>4.15</td>
<td>Compression and decompression times for PRE-PAIR (WSJ20)</td>
<td>104</td>
</tr>
<tr>
<td>4.16</td>
<td>The seven streams by PRE-PAIR and RE-PAIR</td>
<td>106</td>
</tr>
<tr>
<td>5.1</td>
<td>Statistics from block-wise character-based RE-PAIR (WSJ508)</td>
<td>112</td>
</tr>
<tr>
<td>5.2</td>
<td>Final phrase hierarchy of Figure 5.3 in sorted order</td>
<td>123</td>
</tr>
<tr>
<td>5.3</td>
<td>Binary searching a phrase hierarchy using extended symbols</td>
<td>126</td>
</tr>
<tr>
<td>5.4</td>
<td>Additional phrase hierarchy symbol links</td>
<td>128</td>
</tr>
<tr>
<td>5.5</td>
<td>Compression levels using GZIP, BZIP2, and PPMD (NEWS)</td>
<td>143</td>
</tr>
<tr>
<td>5.6</td>
<td>Statistics for RE-MERGE with character-based RE-PAIR (WSJ508)</td>
<td>145</td>
</tr>
<tr>
<td>5.7</td>
<td>Compression levels for RE-MERGE with character-based RE-PAIR (WSJ508)</td>
<td>145</td>
</tr>
<tr>
<td>5.8</td>
<td>Compression times for RE-MERGE with character-based RE-PAIR (WSJ508)</td>
<td>147</td>
</tr>
<tr>
<td>5.9</td>
<td>Applying PRE-PAIR for RE-MERGE (NEWS)</td>
<td>147</td>
</tr>
<tr>
<td>5.10</td>
<td>Statistics for RE-MERGE with punctuation-aligned RE-PAIR (NEWS)</td>
<td>148</td>
</tr>
<tr>
<td>5.11</td>
<td>Compression levels for RE-MERGE with punctuation-aligned RE-PAIR (NEWS)</td>
<td>148</td>
</tr>
<tr>
<td>5.12</td>
<td>Compression times for RE-MERGE with punctuation-aligned RE-PAIR (NEWS)</td>
<td>149</td>
</tr>
<tr>
<td>5.13</td>
<td>Statistics for appending text with character-based RE-PAIR (WSJ508)</td>
<td>152</td>
</tr>
<tr>
<td>5.14</td>
<td>Encoding times for appending text with character-based RE-PAIR (WSJ508)</td>
<td>154</td>
</tr>
<tr>
<td>5.15</td>
<td>Decoding times for appending text with character-based RE-PAIR (WSJ508)</td>
<td>154</td>
</tr>
<tr>
<td>6.1</td>
<td>Mapping differences to nibble-aligned codes for RE-VIEW</td>
<td>170</td>
</tr>
<tr>
<td>6.2</td>
<td>Statistics about the modifier streams and reduced word sequence (NEWS)</td>
<td>173</td>
</tr>
<tr>
<td>6.3</td>
<td>Compression effectiveness for the modifier streams (NEWS)</td>
<td>175</td>
</tr>
<tr>
<td>6.4</td>
<td>Compression efficiency for the modifier streams (NEWS)</td>
<td>177</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.5</td>
<td>Compression statistics for the reduced word sequence (NEWS)</td>
<td>179</td>
</tr>
<tr>
<td>6.6</td>
<td>Distribution of symbols for reduced word sequence experiments (NEWS)</td>
<td>181</td>
</tr>
<tr>
<td>6.7</td>
<td>Search efficiency of RE-VIEW16 for frequency groupings (NEWS)</td>
<td>183</td>
</tr>
<tr>
<td>6.8</td>
<td>Search efficiency of RE-VIEW16 for symbol length groupings (NEWS)</td>
<td>185</td>
</tr>
<tr>
<td>6.9</td>
<td>The seven streams by PRE-PAIR and RE-PAIR, updated</td>
<td>187</td>
</tr>
<tr>
<td>7.1</td>
<td>Recall of symbols in the phrase hierarchy (WSJ20)</td>
<td>207</td>
</tr>
<tr>
<td>7.2</td>
<td>The effect case folding and stemming has on the phrase hierarchy (NEWS)</td>
<td>209</td>
</tr>
<tr>
<td>7.3</td>
<td>Comparing the phrase hierarchy against the noun phrases (WSJ20)</td>
<td>215</td>
</tr>
<tr>
<td>7.4</td>
<td>Recall of noun phrases in the phrase hierarchy (WSJ20)</td>
<td>217</td>
</tr>
<tr>
<td>8.1</td>
<td>Breakdown of requests by file type and origin</td>
<td>224</td>
</tr>
<tr>
<td>8.2</td>
<td>Snapshot of a web site, 24 July 2000</td>
<td>224</td>
</tr>
<tr>
<td>8.3</td>
<td>Disk space required and bandwidth saved for each compression strategy</td>
<td>230</td>
</tr>
<tr>
<td>B.1</td>
<td>Steps in PRE-PAIR's version of the Porter stemming algorithm</td>
<td>243</td>
</tr>
<tr>
<td>B.2</td>
<td>Example of reversing stemming</td>
<td>244</td>
</tr>
</tbody>
</table>
List of Algorithms

2.1 LZ77 .................................................. 19
2.2 LZ78 .................................................. 22
3.1 Re-Pair ................................................. 56
5.1 Re-MERGE phase 2a ................................. 121
5.2 Re-MERGE phase 2b ................................. 133
6.1 Indexed SHUFF ....................................... 165
6.2 Re-View ............................................... 171
Chapter 1

Introduction

I find bookstores irresistible. Recently, a new one opened near my home and while this store was no different from others already in the neighbourhood, I ended up spending half an hour wandering through its bookshelves. I didn’t purchase a book that day, but the visit was not wasted. On my next trip, I was in search of a particular title. By then, I knew that it would have the book I wanted, and also its most likely location. I was correct, and ended up purchasing the book with relative ease.

These two separate trips to the bookstore illustrate two methods of approaching a complex object: browsing and searching. The Australian Concise Oxford Dictionary (third edition) defines browsing as, “[to] read desultorily (going constantly from one subject to another, especially in a half-hearted way)”. While this definition resembles my first visit to the bookstore, it is not a complete description of my action. Without a doubt, if I constantly went from one bookshelf to another, I would have received some suspicious looks from the staff. Moreover, browsing a store, whether it is a bookstore or a clothing store, may not have anything to do with reading. Intuitively, browsing can also be defined as casually looking at various facets of an object in order to get an overall impression of what it represents. Surprisingly, this definition resembles the one for “browser”, also from the same dictionary: “(Computing) a software program that enables one to search for and access documents on the World Wide Web”. In contrast to browsing, searching requires a concise specification of what is being sought; for a book, this could be an author’s name, a book title, or a subject area.
Introduction

As a second example of searching and browsing, Figure 1.1 depicts the two main ways of approaching a university library. Along the top of the figure are the browsing options available to students on their first visit. Some libraries provide a building directory, either as a sign at the entrance, or in the form of a pamphlet. The directory may contain a map, as is the case in the figure, with words indicating the different sections of the library. After looking at the directory, students may choose to browse the bookshelves until finally arriving at a book that piques their interest. Alternatively, the student may arrive at the library because a book is required for an assignment. The assignment’s guidelines usually provide information such as a catalogue number or a book title. This information can be entered as a query into a library catalogue interface, similar to the one shown at the bottom of the figure. A set of results is returned by the system in the form of a list of books that satisfy the query. The list may lead the student to the book required, or if that particular book is not available, the student may choose to browse the bookshelf where the subject is covered.

Note that the figure has been somewhat simplified. For example, after the catalogue at the University of Melbourne\(^1\) returns a set of results, a library patron may select a book followed by the option labelled, “Show Items Nearby on Shelf“. Books that are physically near the selected book on the bookshelf are displayed to the user in a way that resembles browsing. Searching and browsing are so tightly entwined, we may not realise which action is being performed. A student entering a library, with or without an assignment to do, may jump back and forth between these two methods during a single library visit without noticing. Moreover, Figure 1.1 implies that once a book has been found, the student’s retrieval task has been completed. In fact, both browsing and searching occur at multiple levels, and continues after the book of interest has been located. The table of contents and index both provide support for searching, while any part of the book, from the front cover to the brief description on the back, offer chances for browsing.

As for the library’s directory, interesting problems are faced when designing it. Including too little or too much information would make the directory either unhelpful, or

\(^1\)http://cat.lib.unimelb.edu.au/
too cumbersome for library customers. A balancing act is required between the needs for information, and the need to avoid information overload.

The main investigation in this thesis is to explore the compromises required to permit both retrieval and compression of documents when stored in a computer. As with the library’s directory, these tasks are in tension. Documents occupy disk space and network bandwidth which need to be minimised for storage or transmission efficiency. Compression is applied to achieve this. However, after a certain point, the document becomes too compressed to offer any information for retrieval. The only solution, then, is to completely decompress the document. Retrieval and compression are elaborated in the following sections.
1.1 Information Retrieval

Information retrieval (IR) is the study of methods for obtaining information from a repository. The repository may contain a single document, or a collection of documents, and can be managed in a variety of ways. The quality of the documents stored might range from a hastily-written memo, to important contracts, and possess a range of requirements for authentication and archiving.

The World Wide Web [Berners-Lee, 1996] is perhaps the largest document collection in existence at the moment, built without any rules for publication. Since almost anyone can create a web page (and probably has!), the quality of the information varies from outright lies to high-quality journal articles. Despite this anarchic nature, the Web is nevertheless an extraordinarily rich source of information.

Another method of obtaining information is through digital libraries [Witten and Bainbridge, 2002], which vary in size, but are typically much smaller than the Web. Just as in a physical library, a digital library is governed by guidelines which specify an acquisition policy, a method of ensuring longevity, and a comprehensive index. Because of their low cost of duplication (for example, on CD), digital libraries of this form offer the possibility of becoming an enormously useful resource in societies without access to conventional libraries.

Both of these examples are applications of IR techniques, and the document repository that lies behind an IR system can vary greatly in size and in the method with which it has been assembled. Perhaps the most common type of repository is a homogeneous collection of documents that might even be physically stored as a single large file. This is the model used in this thesis, a structure that is elaborated on in Chapter 2.

The interactions between a user and a document repository are illustrated in Figure 1.2. At first, the user has an information need, which is entered as a query via the retrieval system’s interface. The interface sends the query to the retrieval system for processing. The data repository is accessed and the results from the query are returned by the system for display within the interface seen by the user. The result is usually a summary of the document, or document sections, which satisfy the query. But the actual relevance of the
1.1 Information Retrieval

![Diagram of information flow between user, interface, and IR system]

Figure 1.2: Information flow between a user and a repository maintained by an IR system.

result to the original information need of the user is something that only the user can judge. To assist the user to make that decision, the IR system may employ heuristics that attempt to highlight the most likely relevant results, for example, by sorting. The suitability of the results depends on the design of the system, as well as how the information need was represented in the interface.

The quality of a retrieval system is measured in terms of efficiency and effectiveness [Witten et al., 1999]. The efficiency of a retrieval system refers to the time and hardware resources required to provide a response to the user. Effectiveness refers to the accuracy, or correctness, of the result.

If the query is in the form of a single word or a precise phrase, a string matching algorithm such as Boyer-Moore [Boyer and Moore, 1977] can be used to locate sections of documents where that word or phrase appears. The search is performed sequentially from the beginning of each document. The GREP system [Kernighan and McIlroy, 1979] for UNIX operating systems can be used to search across whole directories of documents, and the “search” feature in a word processor is an example of sequential searching within a document.

On the other hand, if a collection of documents needs to be accessed with a variety of queries, information retrieval techniques are required. Data structures should be created to provide efficient, repeated searching. Some facility may also be required to store the data in the repository, while providing simultaneous access from multiple users. Examples of such systems include an encyclopedia CD ROM with a built-in “find” dialog box, or a
Introduction

search engine for the Web. The purpose of a search engine is to assist users in locating web pages which satisfy their information need. In this scenario, the query is a set of words (or terms) and the result is a list of web pages which the system believes satisfies the query.

Traditional IR systems provide two main methods of querying. A retrieval system that supports Boolean queries retrieves documents based on the presence or absence of query terms. Alternatively, a ranked query system returns a list of results, ordered according to a heuristic similarity score that is based on the number of query terms that appear in documents, and the frequency with which they appear in that document and in the collection as a whole. Instead of sequentially searching the entire repository for each query term given, these two retrieval techniques are usually implemented using data structures which index the collection. More detailed descriptions of retrieval systems and their implementation can be found in the books of Frakes and Baeza-Yates [1992], Korfhage [1997], Baeza-Yates and Ribeiro-Neto [1999], and Witten et al. [1999]. Recent books which provide more information about digital libraries include Lesk [1997], Witten and Bainbridge [2002] and Borgman [2003].

As with the bookstore and the university library, searching alone may be insufficient to meet all needs, and providing the user with multiple tools can only aid the searching experience. One additional tool that helps is browsing [Palay and Fox, 1981, Baeza-Yates and Ribeiro-Neto, 1999]. In a document repository, the lexicon is an ideal candidate for browsing [Kowalski, 1997]. The lexicon is the list of all words found in the collection. An extension to lexicon browsing is phrase browsing, where the dictionary used for perusal consists of phrases in the repository, rather than just individual words.

The extraction of phrases from a document can be performed in many ways. The simplest, yet most tedious approach, is to manually select key phrases. If computers are used, then two main methods exist. First, phrases can be formed statistically, based on frequency of word combinations. Second, phrases can be chosen through semantic methods with the help of natural language processing techniques. Wacholder and Nevill-Manning [2001] provide an overview of these and other methods of phrase selection and phrase
1.2 Text Compression

Text compression is the process of identifying and representing the redundant information in a document in a more economical way, so as to minimise the storage space or transmission cost and time.

The desire to find ways to minimise the amount of writing required to speed message transmissions has been around for centuries in the form of abbreviations and symbols. Shorthand, the art of rapid writing using abbreviations, has been used as far back as 2,000 years ago when debates were recorded within the Roman Senate during the Roman Empire. Around that time, Tiro developed a method of shorthand, and is believed to have invented the ampersand (&) [Kreitzman, 1999], a symbol commonly found today in informal text. Attention was drawn to the symbol during Medieval times when monks used it as a sophisticated substitute for the Latin word, “et”, meaning “and” [Todd, 2001]. The symbol is derived from the ligature of the letters “e” and “t”, as shown in Figure 1.3. Moreover, the phrase “et cetera”, which means “and similar things or people”, is often abbreviated now as “etc.”, but was further shorted to “&c” up until the end of the 19th century, according to Todd. The word “ampersand” comes from the contraction of the expression “and per se = and”, whose literal meaning is: “& by itself is and” [Moore, 1997].

Another symbol found in the English language is the ditto mark ("). The word ditto comes from the Tuscan dialect of Italian, and has been a part of the English language since the seventeenth century [Todd, 2001]. The symbol is used in informal writing as a way to avoid writing the same word or phrase multiple times.

Generally, a fundamental requirement of abbreviations and symbols is to be able to reproduce what they represent – during the Roman Empire, one method was to have...
Introduction

![Diagram of the ampersand in different forms: (a) Pompeian graffiti 79 A.D., (b) Scottish writing 9th century, (c) Humanist minuscule 1453 A.D.]

Figure 1.3: Changes to the ampersand through time. Figures and captions taken from Caflisch [2003].

Multiple people record a dictation and then consolidate the differences afterwards [Pitman, 1891]. In the two millennia since the use of shorthand by the Romans, the compression field has evolved into a highly-studied discipline, primarily because of computers. Manual shorthand is limited to the speed at which text can be spoken. Its purpose is to increase the bandwidth of the person talking, at the cost of writing out the text in full later. Computers have allowed complex methods of compression to be used on large documents at very high speeds for both efficient transmission and storage. Compression with computers is not only fast in comparison, but also less error prone than shorthand. Compression algorithms differ in many ways, including the speed at which they process text and how effectively they can compress a document. Books which cover the different approaches in the field of compression include Storer [1988], Bell et al. [1990], Nelson and Gailly [1996], Sayood [1996], Witten et al. [1999], Salomon [2000], and Moffat and Turpin [2002]. A more detailed discussion of compression is provided in Chapter 2.

1.3 Browsing and Compression

Browsing and compression are two key areas of text management, but serve different purposes. The aim of browsing is to present information about a document in order to help satisfy a user’s information need. The purpose of compression is to minimise the size of a document prior to storage or transmission. These two goals are not necessarily compatible with each other. The more compressed a document is, the more difficult it is to search it without completely reversing the transformation. These two areas of text
management are in tension, and the challenge is to find a useful balance point between them.

The aim of this thesis is to investigate the relationship between compression and retrieval, in the context of a homogeneous document collection, and to develop principles that demonstrate how a compromise between the two can be achieved. The emphasis is on phrase browsing, so a more precise objective of this thesis is to consider how a document can be compressed so that phrases drawn from it can still be browsed, and then used as the basis for the retrieval of documents and document fragments. The system described here builds on a compression mechanism called RE-PAIR [Larsson and Moffat, 2000]. This thesis investigates aspects of the RE-PAIR algorithm that need to be improved in order to provide feasible browsing.

1.4 Structure

This thesis is structured as follows. Chapter 2 presents an overview of compression, beginning with some definitions from information theory, and followed by an explanation of how a compression system is composed of a model, a probability estimation process, and a coder. Algorithms representing these three parts are shown, as well as implementations of the compression systems that are used in this thesis for experimental comparisons. A description of the test data for experiments throughout the thesis is also provided.

The RE-PAIR compression process is explained in Chapter 3. RE-PAIR is a dictionary-based compression algorithm which builds a collection of phrases [Larsson and Moffat, 2000]. Experimental results are presented that validate the analysis of Larsson and Moffat, and demonstrate RE-PAIR's compression efficiency and effectiveness. While previous work has looked at the time and memory space requirements of RE-PAIR in the context of the data structures employed, the work in Chapter 3 extends this analysis by considering problems with the algorithm which become evident when compression is not the only goal. An overview of four proposed changes to the algorithm is given, as well as the reasons for choosing RE-PAIR for phrase browsing, despite the problems noted. The four changes are explained in detail in the four following chapters.
Introduction

The original RE-PAIR algorithm selects phrases based on symbol pair frequencies. In Chapter 4, three new methods of selecting phrases for RE-PAIR are given as alternatives to the one used in the implementation of Larsson and Moffat [2000]. The addition of these alternatives provides better quality phrases for phrase browsing, while ensuring that they are properly aligned on word boundaries. Experimental results show the difference in compression effectiveness and the improvement in the types of phrases available to the user.

The amount of memory used by RE-PAIR limits the size of the documents it can process. The solution employed by Larsson and Moffat [2000] is to partition large documents into blocks and then process each block independently. Unfortunately, this approach separates the dictionary into disjoint components, making it difficult to browse as a whole. Chapter 5 proposes a block merging scheme whose primary goal is to combine the separate dictionaries into a single one. As a secondary goal, the block merging scheme improves the compression effectiveness of RE-PAIR by removing redundancies between blocks. Experimental results are given that show the usefulness of this technique.

The coder accompanying the RE-PAIR algorithm is equally important for efficient browsing, and is the subject of Chapter 6. Since the original intention of RE-PAIR was good compression effectiveness, Larsson and Moffat [2000] selected an entropy coder to couple with the parser. However, a static coder that sacrifices some compression effectiveness for retrieval efficiency has the potential to reduce the amount of waiting time experienced by a user and support searching operations better. An experimental comparison between the new coder proposed and a standard Huffman coder concludes the chapter.

The last enhancement to RE-PAIR is the addition of a phrase browsing interface and a random access decoding tool, described in Chapter 7. While Nevill-Manning et al. [1997] shows how a phrase browsing mechanism can be based on the compression algorithm SEQUITUR [Nevill-Manning and Witten, 1997], inherent differences between the two compression algorithms, coupled with the choices made in the previous chapters yield a system with different capabilities. This chapter describes the tool and the interface which allow
the user to browse phrases and, when a phrase of interest has been found, quickly access the contexts in which the phrase occurs. An analysis of the phrase browser is conducted with respect to efficiency, effectiveness, phrase quality, and related work. The SEQUITUR algorithm is described earlier in Chapter 3.

Chapter 8 considers a problem which is separate from the work covered in previous chapters. Instead of large document collections, this chapter investigates how compression can reduce bandwidth requirements for interactive web browsing. Information contained in web server logs are exploited in order to provide an experimental framework based on the HTML files.

Finally, Chapter 9 concludes this thesis by returning to its main theme. This final chapter looks at how RE-PAIR and the additions to RE-PAIR combine to form a phrase browsing system for homogeneous document collections of up to a gigabyte or more. Future directions for this work are also suggested.

To complete the thesis, Appendix A lists notation and Appendix B describes the Porter stemming algorithm, employed in Chapter 4.
Chapter 2

Text Compression

Compression is the fundamental area of text management which recognises and removes redundancies, and represents the remaining content in a more economical way so that the underlying information is retained. The compressed document is then transmitted or stored for later use. The receiver of the compressed document must reverse the transformation in order to obtain the original. The reduction in document size usually comes at the expense of another factor, such as processing time for the conversion.

This chapter provides some background information on compression, and is structured as follows. Section 2.1 presents fundamental theory which forms the foundation of all compression methods. Section 2.2 gives an overview of compression systems, and how they are divided into its three parts: model, probability estimator, and coder. Section 2.3 describes models and probability estimators in the context of three classes of algorithms prevalent in text compression: dictionary-based, statistical, and block sorting. Section 2.4 gives insight into static and entropy coding techniques. The focus of this chapter shifts to practice in Section 2.5, when some complete compression systems are described. Details of the framework for the experiments performed throughout this thesis are covered in Section 2.6. Finally, this chapter concludes in Section 2.7 with a discussion of the effect compression has on searching.
2.1 Fundamentals in Compression

Some definitions and background about the area of information theory as the foundation for compression are required as a first step. We suppose that a message is created by some source. Whether a message is created for transmission or storage for archival purposes, its recipient is the receiver. The channel is the medium between the source and the receiver, and can be a communications line or a disk drive. Between the source and the channel is the process which performs compression: the compressor. Its counterpart is the decompressor, employed by the receiver to reconstruct the message.

The smallest unit in the message is a symbol. The set of all possible symbols available to the source for building messages is the alphabet, denoted as $\Sigma$. The symbols in the alphabet depend on the source and the type of messages being transmitted. For example, if the source only produces telephone numbers, then an alphabet which includes the 10 digits and the hyphen is sufficient. The size of this alphabet (denoted as $|\Sigma|$) is 11. In this thesis, all messages are encoded using the Latin-1 character set [International Organization for Standardization, 1998], an extension to the ASCII character set [American National Standards Institute, 1997]. The ASCII alphabet consists of 128 symbols, including the 26 letters of the English alphabet in both cases, the 10 digits, and a range of punctuation marks and non-printing characters. The Latin-1 alphabet adds an additional 128 symbols to ASCII, to permit the encoding of text in most western European languages. The symbols in a message encoded with Latin-1 are also called characters.

Each symbol in a message conveys some information to the receiver. The amount of information conveyed depends on the probability of the symbol occurring in the message. Since a highly probable symbol offers little surprise to the receiver, the information content is small. On the other hand, an unlikely symbol has a high information content. If $\alpha$ represents a symbol in the alphabet, then the relationship between its information content, $I(\alpha)$, and its probability, $P(\alpha)$, was defined by Shannon [1948] as:

$$I(\alpha) = - \log_2 P(\alpha).$$  

(2.1)
In Equation 2.1, any base can be selected for the logarithm. However, since compression is usually performed with computers based on the binary counting system, a base of two is a convenient choice, and the information content is measured in units of bits. For example, in a message representing a sequence of throws of a six-sided dice, the probability of any value appearing is 1/6. The information content of each of the 6 events is 

\[ -\log_2(1/6) = 2.58 \text{ bits} \]

If the information content of all of the symbols in some alphabet \( \Sigma \) are averaged, then the zero-order entropy of the source is obtained [Shannon, 1948]:

\[
H = \sum_{\alpha \in \Sigma} P(\alpha) \cdot I(\alpha) = - \sum_{\alpha \in \Sigma} P(\alpha) \cdot \log_2 P(\alpha) .
\]  

(2.2)

If the probabilities are derived from a specified message, rather than the source, then the quantity described in Equation 2.2 is the per-symbol zero-order self-information of the message. Shannon demonstrated that the entropy is the lowest limit (in bits per symbol) with which a message from the source can, on average, be coded.

The definitions for information content and entropy depend on two factors. First, they assume that the probability of the symbols in the alphabet can be obtained. While other definitions of entropy exist, this definition assumes a Markov source that generates the symbols in a message independently and identically distributed. That is, the value of a symbol does not affect the values of neighbouring symbols. Of course, this is not true for the letters that appear in an English message, for example. Despite these requirements, the definition of entropy has served as a guide for work in compression for many decades. A more thorough coverage of information theory can be found in books such as Cover and Thomas [1991].

2.2 Compression Systems

Many types of compression systems exist in order to satisfy a range of needs. A system is lossless if the decoder is able to recreate the original message exactly. A system that does
Text Compression

not achieve this goal is *lossy*. Lossy compression is commonly found in the audio and video compression fields, since changes to the message that go unnoticed by the human senses are considered acceptable. Some work has been conducted in the area of lossy compression for text. Nielsen et al. [1997] considered the lossy compression of HTML documents by transforming them to improve bandwidth, provided they rendered correctly within a web browser. Katajainen et al. [1986] applied a syntax-directed compression scheme to Pascal source code. As a higher priority was given to the program source code, other free-form text such as comments and extraneous whitespaces were modified as necessary. Despite these counterexamples, text compression is usually taken to imply lossless compression, because a missing character or word from a sentence can greatly alter the meaning of the sentence.

Whether a compression system is tailored for audio, video, or text, it can usually be divided into three tasks: modelling, probability estimating, and coding [Rissanen and Langdon, Jr., 1981]. The model receives the message from the source and accumulates knowledge about it. The definition of entropy earlier was based on independently occurring symbols, which is not true for most messages. If the probability a symbol occurs is based on neighbouring symbols, it is during the first phase of the compression system that this relationship is captured. Next, the probability estimator assigns probabilities to symbols in the message to form the model, usually based on the observed frequencies derived from the part of the message processed. Finally, the coder maps the output of the probability estimator to a set of codewords based on the *channel alphabet*, which in this thesis is assumed to be binary.

As shown in Figure 2.1, these three components are situated before (and after) the channel, forming the compressor (and decompressor). The decompressor reverses the operations performed by the compressor to obtain the original message. In a compression system, these three parts may each exist in isolation, or be tightly bounded together, with no apparent division between them.

Each component of the compression system operates *statically*, *semi-statically*, or *adaptively*. A static algorithm makes use of fixed, pre-determined statistics to process
the input. These statistics are presumed to be available at both ends of the channel, so that explicit transmission is unnecessary. The drawback to such a regime is that it is only suitable when the static probabilities match those found in the input. When there is a mismatch, compression effectiveness degrades.

On the other hand, an adaptive algorithm learns about the input throughout the compression process, and typically commences with bland statistics. As compression progresses, these statistics are updated, in an attempt to match those possessed by the message. Like static algorithms, adaptive ones have the advantage of requiring only a single pass. Furthermore, adaptive mechanisms are suitable for any type of message.

Semi-static algorithms encode the message in two passes. The first pass collects information about the message. That knowledge is then applied in the second pass. The primary difference between a static algorithm and a semi-static equivalent is that the information that exists on both sides of the channel in the static approach must be explicitly transmitted by the semi-static one. Thereafter, the process is the same.

Closely related to these three algorithm classifications are the categories on-line and off-line. An on-line algorithm processes a symbol, and then immediately commits it to the channel as a stream of bits before reading in the next symbol. An off-line mechanism has the luxury of examining a significant block of the input before sending anything to the decoder, at the cost of buffering symbols in memory.
Text Compression

<table>
<thead>
<tr>
<th>how much wood could a woodchuck chuck?</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a woodchuck could chuck wood...</td>
</tr>
<tr>
<td>as much wood as a woodchuck would chuck</td>
</tr>
<tr>
<td>if a woodchuck could chuck wood</td>
</tr>
</tbody>
</table>

Figure 2.2: The “Woodchuck” message. A space is indicated by a “.” character. Linefeeds have been inserted to improve readability.

In the next two sections, models and coders are discussed in greater detail through sample algorithms. Coverage of probability estimators is absorbed into the description of models, because of their close relationship.

2.3 Modelling

The model analyses and learns about the message, and in the process, identifies the redundancies that allow the message to be represented in an alternate, yet cost-effective manner. Three different models are considered in this chapter: dictionary-based, statistical, and block sorting.

A short passage has been chosen as the example for most of this thesis. The “Woodchuck” tongue twister from Opie and Opie [1959, pg. 30] is shown in Figure 2.2. All words in the message have been case folded, and all punctuation has been removed. In addition, all whitespace characters have been collapsed into a single space, as indicated in the figure as “.”. While linefeeds are used in the figure to improve display, they are not considered to be part of the message.

Dictionary-Based Models

Dictionary-based models (or macro encoding mechanisms [Storer and Szymanski, 1982]) seek to represent the input message in a more cost effective way by building a dictionary of phrases. Sections of the message are substituted by references to the dictionary’s entries. Compression is achieved if the cost of expressing the dictionary is less than the total number of symbols it replaces.
Algorithm 2.1: Algorithm for LZ77.

\begin{algorithm}
\begin{algorithmic}
  \State \textbf{input} : input message $\mathcal{M}$, window size $N$
  \State \textbf{output} : encoded message $\mathcal{M}'$
  \State $\text{prev\_offset} \leftarrow 0$; $\text{offset} \leftarrow 0$
  \State $i \leftarrow 0$; $\omega \leftarrow \lambda$
  \For{$i < |\mathcal{M}|$}
    \State $\text{offset} \leftarrow \text{findLongestMatch}(\omega, \mathcal{M}[i])$
    \If{$\text{offset} = 0$}
      \State output \{$\text{prev\_offset}, |\omega| - 1, \mathcal{M}[i]\$} to $\mathcal{M}'$
      \State $\text{prev\_offset} \leftarrow 0$
      \State $\omega \leftarrow \lambda$
    \Else
      \State $\text{prev\_offset} \leftarrow \text{offset}$
    \EndIf
    \State $i \leftarrow i + 1$
  \EndFor
  \State \Return{$(\mathcal{M}')$}
\end{algorithmic}
\end{algorithm}

A static dictionary-based model would hard-code the phrases in the compressor and the decompressor. The dictionary can also be built semi-statically or adaptively so that each compressed message has an associated dictionary. In this case, either the dictionary must be explicitly sent to the decompressor, or enough information must be transmitted for the decompressor to reconstruct the dictionary. The overall effectiveness of a dictionary-based model depends on the frequency with which dictionary phrases occur, and their lengths.

This section focuses on two dictionary-based models, both of which operate adaptively and on-line. The first is the LZ77 algorithm by Ziv and Lempel [1977], which creates a dictionary by maintaining a window of recently seen symbols. Each symbol in the window is considered to be the beginning of a phrase of arbitrary length. Sequences (or strings) of symbols in the message are replaced by phrase references.

Algorithm 2.1 presents the algorithm for LZ77. The input and output messages are represented as $\mathcal{M}$ and $\mathcal{M}'$, respectively. The size of the window is denoted as $N$. As each symbol is read, it is appended to the string $\omega$, which is initialised to the empty string $\lambda$ at the beginning of the algorithm on line 2. On line 5, a search for $\omega$ is performed on the dictionary of phrases for the set of strings that have $\omega$ as a prefix. A match is indicated
Text Compression

by a distance offset backwards, counting from the end of the window. When \( \omega \) cannot be
found, an offset of 0 is returned from the search, and the string \( \omega \) is encoded as a 3-tuple
of the form \( \langle d, l, s \rangle \). The location in the window of the last successful match is specified
by \( d \), while the length of this match (one less than the length of \( \omega \)) is specified by \( l \). The
last symbol in \( \omega \) that caused the mismatch is transmitted directly as \( s \). As a special case,
when the string \( \omega \) only contains one symbol, \( \alpha \), the novel symbol is replaced with the
tuple: \( \langle 0, 0, \alpha \rangle \).

The size of the window is restricted to some value \( N \), and as more of the message is
processed, symbols at the beginning of the window are removed so that the window is kept
recent. The eradication of old symbols and the addition of new ones give the illusion of a
window sliding over the message. So, mechanisms based on LZ77 are sometimes referred
to as “sliding window” compression schemes. For efficiency reasons, \( N \) is usually chosen
as a power of 2.

Table 2.1 illustrates the LZ77 algorithm with an example. The sample text is formed
from part of the “Woodchuck” message, “woodchuck chuck if”. Each line in the figure
indicates one 3-tuple. The dictionary window is indicated by a shaded box, while \( \omega \) is
placed in a framed box. The window is \( N = 8 \) symbols in length. In row (a) of the
table, the window is initially empty, and the tuple \( \langle 0, 0, w \rangle \) is transmitted, to get the first
character through. Later, in row (c), the second occurrence of the letter “o” creates an \( \omega 
\) 
two characters in size. At the second to last step in the table, \( \omega \) is 7 characters in length
and the longest match is 6 characters from the end of the window. At this stage, the first
two characters in the message, “wo”, have departed from the window.

In terms of memory, the LZ77 algorithm requires at least space proportional to \( N \) in
order to hold the sliding window. However, since searching for \( \omega \) in the dictionary can be
the most costly operation in the algorithm, additional data structures are used to improve
efficiency. Some basic data structures include sorted lists and binary search trees. Other
possible mechanisms include tries (also called multi-way tries [Sedgewick, 1997]) and hash
tables. A trie is a tree where each node has an outdegree equal to \( |\Sigma| \). When searching for
a string \( \omega \), reading each character results in one transition from the current node to one

20
2.3 Modelling

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(b)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(c)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(d)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(e)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(f)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(g)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(h)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(i)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
<tr>
<td>(j)</td>
<td>\texttt{woodchuck, chuck, if}</td>
</tr>
</tbody>
</table>

Table 2.1: Dictionary-based compression using the LZ77 algorithm. Each row illustrates the string \( \omega \) being encoded and its associated 3-tuple. On the right, the shaded region is the current window, which grows until a fixed size and then "slides" across the text. The unshaded region is the section of text being processed \( \omega \).

of its \( |\Sigma| \) child nodes. A hash table can be implemented so that the first few characters of each phrase are used to calculate a position in the table. The data structure chosen must also take into account the fact that symbols leave the window as well as join it. So, phrases are continually added and removed from the data structure.

In comparison, the decoder is simpler than the compressor. While a sliding window must be maintained, string searching is not required. Each tuple in the compressed message indicates explicitly where in the window to copy phrases from, and the number of symbols to copy. Hence, decoding is extremely quick. In fact, the fast decoding time is a trait shared by most dictionary-based algorithms. Reconstructing the message by directly copying strings from the dictionary is attractive from a pragmatic point of view.

Some variations on the LZ77 algorithm include LZR and LZSS. The LZR scheme [Rodeh et al., 1981] removes the size of the sliding window so that all of the message seen so far is used as phrases. Bell [1986] designed LZSS to remove the restriction on constant tuple sizes by mixing the output stream with characters and back pointers. A bit flag was used to indicate whether a literal character or a back pointer exists in the stream. This work was taken further by Fenwick [1993], who investigated other lengths of bit flags.

The second dictionary-based algorithm discussed in this chapter is the LZ78 algorithm
Text Compression

Algorithm 2.2: Algorithm for LZ78.

\[
\text{input} \quad : \text{input message } M \\
\text{output} \quad : \text{encoded message } M'
\]

1. \( \text{prev\_position} \leftarrow 0; \text{position} \leftarrow 0 \)
2. \( i \leftarrow 0; \omega \leftarrow \lambda \)
3. for \( i < |M| \) do
   4. \( \text{appendString}(\omega, M[i]) \)
   5. \( \text{position} \leftarrow \text{searchDictionary}(\omega, D) \)
   6. if \( \text{position} = 0 \) then
      7. \( \text{output } \langle \text{prev\_position}, M[i] \rangle \text{ to } M' \)
      8. \( \text{add } \omega \text{ to } D \)
      9. \( \text{prev\_position} \leftarrow 0 \)
      10. \( \omega \leftarrow \lambda \)
   11. else
      12. \( \text{prev\_position} \leftarrow \text{position} \)
      13. \( i \leftarrow i + 1 \)
4. return \((M')\)

of Ziv and Lempel [1978]. Like LZ77, LZ78 is adaptive, operates on-line, and replaces
sections of the message with pointers to the dictionary. However, rather than using a
sliding window of symbols as a phrase book, an explicit dictionary is built. Algorithm 2.2
lists the algorithm for LZ78. As before, symbols are read one at a time and appended to
the string \( \omega \). If \( \omega \) exists in the dictionary \( D \), then the next input symbol is considered.
But if \( \omega \) cannot be found, then it is replaced with a 2-tuple, \( \langle p, s \rangle \). The last successful
match for a phrase (that is, \( \omega \) without its last character) is specified by \( p \) as a position in
the dictionary. The last symbol in \( \omega \) is encoded explicitly as a literal, \( s \). Whenever a new
phrase has been found, it is added to the dictionary on line 8 of the algorithm. Dictionary
entries are indexed from 1 so that a new symbol, \( \alpha \), is encoded as \( \langle 0, \alpha \rangle \). The dictionary
is initially empty.

A sample application of the LZ78 algorithm on the same message as Table 2.1 is shown
in Table 2.2. Each row in the table shows a section of text being encoded as a 2-tuple
(indicated by the column marked “Tuple”). The position in the dictionary of each new
string is shown in the next column, followed by the progress of the compression. The
shaded region represents text that has already been seen, while the framed box draws
2.3 Modelling

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Current dictionary</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) (0, w)</td>
<td>1 → w</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(b) (0, o)</td>
<td>2 → o</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(c) (2, d)</td>
<td>3 → od</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(d) (0, c)</td>
<td>4 → c</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(e) (0, h)</td>
<td>5 → h</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(f) (0, u)</td>
<td>6 → u</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(g) (4, k)</td>
<td>7 → ck</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(h) (0, w)</td>
<td>8 → w</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(i) (4, h)</td>
<td>9 → ch</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(j) (6, c)</td>
<td>10 → uc</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(k) (0, k)</td>
<td>11 → k</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(l) (8, i)</td>
<td>12 → i</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
<tr>
<td>(m) (0, f)</td>
<td>13 → f</td>
<td>\texttt{\texttt{woodchuck_chuck_if}}</td>
</tr>
</tbody>
</table>

Table 2.2: Dictionary-based compression using the LZ78 algorithm. Each row illustrates the string \( \omega \) being encoded, with the generated 2-tuple in the first column. The position in the dictionary and the equivalent string of the new tuple is shown in the second column. At the end of each row is the progress of the algorithm. The shaded region indicates text that has already been processed, while the unshaded region is the section of text being processed (\( \omega \)). Dictionary strings grow one character longer at each step.

attention to the string \( \omega \). Due to the short example, the message is encoded only as strings of length 1 or 2. Since the dictionary gradually grows, the repetition of the string "chuck\_" is not captured within the text shown, and more pointers are required than the equivalent LZ77 parsing. Also, in row (k), note that the letter "k" is encoded as a literal, even though it is the second time it has been seen. The first time it was encountered, it was part of a string of length 2.

In comparison to the LZ77 algorithm, LZ78 searches for phrases in an explicit dictionary, rather than a sliding window of phrases. The relevance of the dictionary to the text being compressed is no longer limited by a window size. Even so, in practice, a limitation on memory usage is imposed on LZ78 so that the amount of space is bounded. Also, while phrases are explicitly added into \( D \), note that the addition of a phrase does not give assurance that it will be used.

The same data structures described for LZ77 can also be employed by LZ78 for main-
Text Compression

taining phrases. But since phrases are never deleted from the dictionary, one complication inherent in LZ77 is eliminated by LZ78. The LZ78 algorithm, though, requires the dictionary to be maintained by both the decompressor and the compressor. As a result, decoding with LZ78 is usually slower than LZ77.

In LZ78 implementations, several approaches exist when dealing with a full dictionary. First, the dictionary can be deleted, and re-constructed from scratch. Second, a least recently used policy can be employed, so that the oldest phrases are removed to make space for new ones [Tischer, 1987]. Alternatively, if every symbol in the alphabet is used to initialise the dictionary (such as the case with LZW by Welch [1984]), then it can be kept static for the remainder of the message. Because of this, the output from the LZW algorithm only contains pointers to phrases. Combinations of these three policies are possible; for example LZW rebuilds the dictionary if compression degrades significantly.

Statistical Models

Instead of building a dictionary, statistical models directly estimate the probability for each symbol in the message. As an example, suppose symbols are words and the following sentence fragment has to be completed with a word, “I purchased a book from the ...”. Many plausible guesses exist, but most of them would be sources of books, such as “bookstore” and “Internet”. This conclusion can be drawn from the information provided by the preceding words.

Generalising the problem to symbols, rather than words, statistical models combine with probability estimation modules in order to gather statistics and provide estimates for each symbol as it is encountered. Restricting the context to the preceding symbols permits messages to be processed in a single pass. An order-\(K\) context uses the previous \(K\) symbols to predict the current symbol. By definition, an order-0 context does not use any preceding symbols, while an order of minus-one assigns the same probability to every symbol. The Latin-1 standard can be thought of as a very simple compression system based on a fixed minus-one order context. An order-1 statistical model predicts each symbol in the context of one immediately preceding symbol. Intuitively, the higher the
2.3 Modelling

order, the more accurately the model is able to predict symbols. However, if the size of the alphabet is $|\Sigma|$, then an increase in the context by one symbol results in an increase in memory usage by a factor of as much as $|\Sigma|$, and in general, for an order-$K$ context, the amount of memory required is potentially proportional to $|\Sigma|^{(K+1)}$.

The Prediction by Partial Matching (PPM) algorithm [Cleary and Witten, 1984] is one example of a statistical model. In order to conserve memory, instead of creating all possible contexts initially, PPM records contexts as they are encountered in the message. If a symbol cannot be predicted in the current context, then the algorithm reduces the context by one symbol and tries again, until a context can be found.

Implementations of the PPM algorithm are typically parameterised in terms of the maximum order $K$. The encoding of a symbol $\alpha$ is first attempted using the previous $k = K$ symbols. If $\alpha$ has not occurred in this context, then a special escape symbol is transmitted to indicate that a context of length $k - 1$ should be attempted. This is repeated until $\alpha$ can be estimated based on a previously seen context, or when $k = -1$ and the symbol is encoded as a literal.

Several variations on the PPM algorithm exist, differing primarily in the probability assigned to the escape symbol. The probability given to the escape symbol is related to the zero frequency problem [Witten and Bell, 1991]. The zero frequency problem refers to the dilemma of assigning a probability to an event that has not occurred. Since there is a chance it could occur later, novel events that have not yet occurred must still be assigned non-zero probabilities.

With respect to PPM, the probability of the escape symbol is an instance of the zero frequency problem since the escape symbol is emitted when a symbol has not occurred in the current context. Several methods related to this problem have been explored. Method A of PPM increases the count of each context by 1 for the escape probability. Method B subtracts 1 from the counts of every context so that an event is considered to have occurred only if has happened twice. The remaining counts are allocated to the escape symbol. Methods A and B were described by Cleary and Witten [1984]. Method C [Moffat, 1990] assigns a probability of $d/(n + d)$, where $d$ is the number of distinct
symbols seen in the current context, and $n$ is the total number of symbols encountered in that context. Method D [Howard, 1993], estimates the probability of the escape symbol as $d/(2n)$. Implementations of PPM which make use of the latter two methods generally achieve better compression effectiveness than variants using methods A and B [Witten et al., 1999]. Cleary and Teahan [1997] looked at PPM with an unbounded context.

An example of the PPM algorithm is shown in Table 2.3. The sample message is again “woodchuck.chuck.if”. A row in the table represents the processing of a character in the message. The message is in the last column, with the shaded region representing text that has already been seen. The framed box indicates the character under consideration. In this example, the maximum order is fixed at $K = 2$. So, an attempt is made to first predict each character using the two immediately preceding characters. The columns labelled “Order” and “Context” indicate the order and context that is finally used to predict the current character. Novel characters are always encoded using a −1 order model. In the example, four of the letters in the second occurrence of “chuck” can be predicted using a second order model.

The amount of memory required by the PPM algorithm is higher than dictionary-based algorithms, due to the number of contexts that have to be maintained. Typical implementations use several megabytes of memory for data structures. Bell et al. [1990] showed how an implementation of PPM would use a trie of context nodes, with each node representing a unique context of symbols. Nodes are interlinked with vine pointers so that when a shorter context is needed, it can be reached in $O(1)$ time. Even so, PPM tends to execute slower for both encoding and decoding in comparison to LZ77 and LZ78 because of the overhead of maintaining the context tree, and the fact that PPM operates on a per symbol basis. It is widely accepted that the combination of PPM with an arithmetic coder (to be discussed below), is one of the best general-purpose compression systems\footnote{The Canterbury Corpus web site at \url{http://corpus.canterbury.ac.nz} compares a wide range of compression systems.}. Bell et al. [1990] summarise work by others (including Shannon [1948], Rissanen and Langdon, Jr. [1981], and Bell and Witten [1987]), and show that an equivalent statistical model
### Table 2.3: Statistical Modelling using PPM

<table>
<thead>
<tr>
<th>Order</th>
<th>Context</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(b)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(c)</td>
<td>0</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(d)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(e)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(f)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(g)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(h)</td>
<td>0</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(i)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(j)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(k)</td>
<td>0</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(l)</td>
<td>1</td>
<td>c woodchuck_chuck_if</td>
</tr>
<tr>
<td>(m)</td>
<td>2</td>
<td>ch woodchuck_chuck_if</td>
</tr>
<tr>
<td>(n)</td>
<td>2</td>
<td>hu woodchuck_chuck_if</td>
</tr>
<tr>
<td>(o)</td>
<td>2</td>
<td>uc woodchuck_chuck_if</td>
</tr>
<tr>
<td>(p)</td>
<td>2</td>
<td>ck woodchuck_chuck_if</td>
</tr>
<tr>
<td>(q)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
<tr>
<td>(r)</td>
<td>-1</td>
<td>woodchuck_chuck_if</td>
</tr>
</tbody>
</table>

Each row indicates the order and context used to predict the current character. The progress of the algorithm is presented at the end of each row, with the current character placed in a box. The shaded region indicates the characters that have been processed.

exists for any dictionary-based algorithm which selects phrases in a greedy fashion. This relationship was further discussed by Bell and Witten [1994]. So, a greedy dictionary-based model cannot outperform a statistical one.

### Block Sorting

The last modelling technique considered in this chapter is the block sorting mechanism devised by Burrows and Wheeler [1994]. Also called the Burrows-Wheeler transform (or BWT), the algorithm differs from dictionary-based and statistical ones in that it applies a transformation to the message to improve compression. A subsequent move-to-front mechanism is invoked in order to underscore the locality created by the transformation, and eventually, a zero-order coder is applied.

The block sorting transform is best explained through an example. Figure 2.3 demon-
strates the algorithm with the string, “woodchuck”\(^2\). The first step in block sorting is to partition the message into blocks so that each one can be processed independently in an off-line manner. For each block of \(n\) symbols, a set of \(n\) strings is created, starting from each symbol in the block. When the end of the string is reached, the string wraps around to the beginning. The \(n\) strings for the example string are presented in Figure 2.3(a). The block of strings is then sorted by comparing from the second-to-last symbol until the beginning of the string, in a reverse lexicographic order. This is shown in Figure 2.3(b). Sorting into context order gives a good indication about what characters appear after a particular conditioning sequence. For example, the first two strings of Figure 2.3(b) show that in this string, the character “c” only forms a context for “h” and “k”. The sequence of last characters of the block of sorted strings is shown with a shaded column in the figure. This string of characters is encoded for transmission. The encoder notifies the decoder as to the location of the symbol at the beginning of the original message. In this example, it is the “w” in the fifth position of the string, as indicated by the asterisk (*).

The decoder receives the permuted string, and sorts it. In Figure 2.3(c), the sorted string on the left is placed next to the received unsorted string, shown to its right. The

---

\(^2\)See other sources, such as Witten et al. [1999, pg. 66], for an example of block sorting using a longer message with more redundancy.
sorted string of received characters is identical to the column of second-to-last characters in Figure 2.3(b). That is, each character in the sorted string forms a one-symbol context for the character to its right. Moreover, in both strings, the relative order of characters which have the same value is preserved. The second "o" in the sorted string (seventh character) is the same character as the second "o" in the unsorted string (last character). One symbol is decoded by examining a character from the received string, then the same character in the sorted version, and finally, the character it forms a context with, to its right, as illustrated in Figure 2.3(c). In this example, the first character in the unsorted string is the "w" in position 5. This character is the same as the "w" at the end of the sorted string, which forms a context for the second "o" in the unsorted string.

The difference between the transmitted string and the original message is that the permuted string has localised character occurrences. In the first two strings of Figure 2.3(b), the characters "h" and "k" appear in the context of "c". Even if the message was longer, only the few character that appear after a "c" would cluster in this region.

To exploit the localised clustering, a move-to-front (MTF) transformation is then applied. The transformation translates each symbol in the string to a number that indicates how many distinct symbols have been observed since this one was last seen. A symbol that was recently seen gets translated into a small value. In a cluster of symbols where there is little variation, sequences of small numbers result. In the end, the permuted string becomes a list of numbers with a skewed distribution of mainly small values, and it is that sequence that is passed to a coder. If the seven distinct characters in "woodchuck" are initially ordered alphabetically, then the MTF values for the permuted string is \{3,4,3,6,7,7,5,3\}. If the original message was longer, then clusters of symbols in the permuted sequence would generate runs of small MTF numbers. As an alternative, Wirth and Moffat [2001] compressed the output from block sorting directly using other schemes, without applying the MTF transform.

Generally, systems which implement block sorting with MTF yield compression ratios close to PPM due to their exploitation of symbol contexts [Cleary and Teahan, 1997]. As the most time consuming operation in BWT is string sorting, others have considered ways
Text Compression

of improving this phase of the algorithm [Sadakane, 1998, Seward, 2000, 2001]. Several other authors have explored refinements to improve compression effectiveness [Deorowicz, 2000, Fenwick, 1996a,b].

2.4 Coders

The model and probability estimator identifies the redundant information in the message and produces a reduced message. This reduced message must then be coded to achieve the desired transmission across the channel. Based on the description of models presented earlier in this chapter, the reduced message may contain components of a 3-tuple from a LZ77 implementation, a stream of symbols indicating which PPM contexts were chosen, or MTF values for a message permuted with block sorting.

The coder obtains the reduced message, the set of distinct symbols in the reduced message \( D \), and the probabilities with which these symbols occur \( P \). From these three pieces of information, the coder creates a set of codewords \( C \) which map each symbol in the reduced message to a string of letters in the channel alphabet. Compression has been achieved if the set of codewords (collectively called codes) chosen produce an output message shorter than the original message. Furthermore, if the codes are selected so that no codeword is a prefix of any other codeword, then the codes are prefix-free. Prefix-free codes ensure that the compressed message is uniquely decodable, without the need to buffer codewords.

Two classes of coders are presented next. Methods in the first class assign codewords statically, based solely on a symbol's value within the alphabet. The second class of coders are entropy coders, which use the supplied symbol probabilities to produce a compressed message whose length is intended to be close to the self-information of the reduced message.

Static Coders

Static coders map each symbol in the reduced message to a particular codeword, without acknowledging the probabilities with which the symbols occur. The simplest scheme is a
2.4 Coders

<table>
<thead>
<tr>
<th>$x$</th>
<th>unary</th>
<th>binary</th>
<th>gamma</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>001</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>010</td>
<td>101</td>
<td>1001</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>011</td>
<td>1100</td>
<td>10100</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>100</td>
<td>11001</td>
<td>10101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
<td>11010</td>
<td>10110</td>
</tr>
<tr>
<td>7</td>
<td>1100</td>
<td>110</td>
<td>11011</td>
<td>10111</td>
</tr>
<tr>
<td>8</td>
<td>1110</td>
<td>111</td>
<td>110000</td>
<td>1100000</td>
</tr>
<tr>
<td>9</td>
<td>11110</td>
<td>1111</td>
<td>110001</td>
<td>1100001</td>
</tr>
</tbody>
</table>

Table 2.4: Some sample static codes for the values 1 to 9. The prefix of the gamma and delta codes are shaded in grey.

Unary code, which encodes the symbol $x$ as a codeword made up of $x - 1$ one-bits, and terminated by a zero bit. Unary codes are biased toward smaller values of $x$, since the codeword for $x$ is exactly $x$ bits in length.

Another straightforward coding mechanism is binary, which allocates a codeword of length $\lceil \log_2 N \rceil$ bits for each $x$, for some upper limit $N$. Unlike unary coding, binary coding assumes a uniform distribution of the symbols and a definite range. Some examples of binary coding are the ASCII and Latin-1 character sets, which encode each symbol in 7 and 8 bits, respectively. The second and third columns of Table 2.4 list the unary and binary codes for the values 1 to 9, respectively. The binary codes in this table are based on an upper limit of $N = 8$. Because of this, no binary code is possible for $x = 9$.

The fourth column of the table shows the codewords produced by gamma coding [Elias, 1975]. Gamma codes offer a compromise between the skewed distribution assumed by unary coding, and the uniform distribution of binary coding. A gamma code has two parts, a prefix and a suffix. The prefix, shown shaded in the fourth column, uses a unary code to specify the binary length of the symbol. The suffix part specifies the exact symbol in binary within that order of magnitude. For the symbol $x$, the prefix specifies $1 + \lceil \log_2 x \rceil$ in $1 + \lfloor \log_2 x \rfloor$ bits, while the suffix indicates $x - 2^{\lfloor \log_2 x \rfloor}$ in $\lceil \log_2 x \rceil$ bits. The total length of the gamma code for $x$ is thus $1 + 2^{\lceil \log_2 x \rceil}$ bits. If the prefix is coded using a gamma code, and the suffix in binary, then the delta codes result, also described by Elias. The last column of Table 2.4 presents some delta codes, with the prefix again highlighted. The
Text Compression

length of a delta code for a symbol \( x \) is \( 1 + 2\left(\log_2 \log_2 2x\right) + \log_2 x \) bits.

There are two benefits of using Elias codes instead of unary and binary codes. In comparison to unary codes, Elias codes are shorter for all but a very small number of values of \( x \). And while the length of binary codes are even shorter, binary codes are assigned based on a known range, which might not be a reasonable restriction. On the other hand, Elias codes and unary codes can represent arbitrarily large values. All four codes are prefix-free.

Huffman Coding

Entropy coders assign codes derived from symbol probabilities that are calculated directly from symbol frequencies in the reduced message, or through other means. Either way, they are assumed to be true for the symbols being processed. Entropy coders that decide on codewords based on the observed frequency of symbols in the reduced message can do so using either semi-static or adaptive mechanisms. Two entropy coding regimes are considered in this section: semi-static Huffman coding, and adaptive arithmetic coding.

Huffman coding is a famous entropy coding technique [Huffman, 1952]. This algorithm assigns codewords so that frequent symbols obtain shorter codewords than rare symbols. The Huffman algorithm calculates codewords incrementally, by first placing each symbol of the alphabet in a package, and then by repeatedly merging the two packages with the lowest combined probabilities, until only one package remains. Whenever two packages are combined, the probabilities are summed. Every symbol has an associated codeword length which is incremented by one every time the symbol participates in a merge. When the algorithm concludes, the set of lengths are used to assign codewords to each symbol.

Figure 2.4 explains the Huffman algorithm through an example. Let the set of packages be \( D \), the associated probabilities of each symbol in the alphabet be \( P \), and their corresponding codeword lengths \( L \). In the example, the alphabet consists of the four symbols \( D = \{\{w\}, \{x\}, \{y\}, \{z\}\} \), with respective probabilities \( P = \{0.1, 0.2, 0.2, 0.5\} \). The codeword lengths are all initialised to 0 so that, \( L = \{0, 0, 0, 0\} \). First, the two packages with the lowest probabilities are merged in Figure 2.4(b). Since the packages \{x\}
and \( \{y\} \) have the same probabilities, either one can be combined with \( \{w\} \). The selected packages, shown shaded, are merged and their codeword lengths are incremented by 1. Once the package \( \{z\} \), is processed in Figure 2.4(d), the final set of codeword lengths is \( L = \{3,3,2,1\} \).

Next, a set \( C \) of prefix-free codewords is assigned to each symbol based on \( L \) to complete Huffman’s algorithm. One set of codewords that satisfy this condition is \( C = \{000,001,01,1\} \). Note that the assignment of codewords is not unique. If there are \( n \) packages, then there are \( n - 1 \) merge operations. Each merge operation is equivalent to appending a 0 bit to one package and a 1 bit to the other, once \( L \) has been calculated. As the algorithm does not state the bit a package receives, there are \( 2^{n-1} \) sets of Huffman codes. Moreover, as shown in the example, when multiple packages share the same probability, ties are broken arbitrarily.

An alternative method to the Huffman code construction described is the bottom-up method which employs a binary tree [Johnsonbaugh, 1993, Salomon, 2000, Cormen et al., 2001]. Figure 2.5 shows a tree for the codewords \( C \) that were assigned using the lengths \( L \) from the example of Figure 2.4. Symbols are placed at the leaves of the tree. In this graph representation, symbol lengths translate to depths of symbols in the tree, so that each edge signifies a bit in the codeword. A bit stream is decoded by traversing the tree
starting from the root, so that a bit permits one edge transition. When a leaf has been reached, the symbol at that leaf is output and the process begins again from the root. While visualising the decoding of Huffman codes with a tree is helpful, Moffat and Turpin [1997] presents a more efficient method for decoding which replaces tree construction by both the encoder and the decoder with a set of arrays. The primary requirement is that the codes must be canonical. A set of Huffman codes are canonical if they are ordered lexically when sorted by length.

According to the codewords of Figure 2.4, the reduced message “yzzw” would be coded in 7 bits as 0111000. Unfortunately, this bit stream alone is not enough for the decoder to recreate the original message. A prelude is required which indicates the alphabet of the reduced message \( D \). Details about the construction and transmission of the prelude for Huffman coding is described by Turpin and Moffat [2000].

Other variations to the Huffman algorithm include length-limited coding [Larmore and Hirschberg, 1990, Liddell and Moffat, 2002] and adaptive Huffman coding [Vitter, 1987, Knuth, 1985]. In length-limited coding, codewords are assigned so that none of them are longer than \( L_m \) bits, for some upper limit \( L_m \). Adaptive Huffman coding differs from the semi-static paradigm described above in that the probabilities of the symbol are modified while the message is processed. Moreover, like the PPM algorithm described earlier, the zero-frequency problem re-emerges since a probability must be assigned to novel symbols.
2.4 Coders

Figure 2.6: The division of intervals when arithmetic coding is applied to the message “yzzw”, for the four symbol alphabet, \{w, x, y, z\}. The intervals are broken down further with each symbol encountered.

Arithmetic Coding

All of the coding techniques described so far have one trait in common. Regardless of the value of a symbol, or its probability in the message, the symbol must be coded in at least one bit. Arithmetic coding by Rissanen [1976] and Witten et al. [1987], and later revised by Moffat et al. [1998], eliminates this restriction by coding sequences of symbols at a time.

Figure 2.6 explains the algorithm through the sample message “yzzw”, using the symbol probabilities used earlier. At first, the interval \([0, 1)\) is divided according to the cumulative probabilities of \(D\). As each symbol in the message is encountered, the section corresponding to the symbol is further divided. In the example, after “y” has been processed, the interval \([0.3, 0.5)\) is divided into four parts based on the symbol probabilities. Once the last symbol is read, the interval \([0.45, 0.455)\) has been isolated. In binary, with the decimal removed, this interval is \([0111001, 0111010001)\). The message can be encoded by specifying any number within this interval. One sequence that meets this requirement is 011101 (or, 0.453125).

As with Huffman coding, the prelude must be sent to the decoder. Furthermore, in order for the decoder to mimic the encoder, the probabilities of the symbols need to be
transmitted as well. Then, the decoder operates by dividing and selecting intervals in the same order as the encoder. Several other simplifications have been made. For example, an actual implementation would use integers instead of real numbers so as to avoid any rounding error. Also, intervals must be scaled when they become too narrow. Moffat and Turpin [2002] provides a detailed discussion about the implementation issues associated with arithmetic coding.

Generally, arithmetic coding performs better than other coding techniques for highly skewed probabilities – especially when the self-information of any of the symbols is less than one bit. In fact, the compression effectiveness attained by arithmetic coding is better able to approach the self-information of the message than any other coding technique. However, in comparison to other coding methods, arithmetic coding is relatively slow for both encoding and decoding, since an interval has to be divided for each symbol in the message.

2.5 Compression in Practice

Three compression systems are explained next, in the context of the algorithm descriptions presented above. These systems, two of which are readily available on the Internet, are used throughout this thesis as baseline comparisons. The three implementations are GZIP, PPMD, and BZIP2.

The dictionary-based system GZIP [Gailly, 1993] employs the DEFLATE algorithm [Deutsch, 1996], which combines a variant of the LZ77 algorithm with Huffman coding. The DEFLATE algorithm partitions a message into blocks, and processes a block at a time for efficiency reasons. The DEFLATE algorithm uses 1-tuples for literal symbols and 2-tuples for references into the dictionary, and neither type of tuple specifies the next symbol. The sliding window size is fixed at 32 KiB3. The DEFLATE algorithm assumes that the message is encoded in bytes, so that literal symbols fall within the range of 0 to 255. Literal symbols and phrase lengths are encoded using one set of Huffman codes,

---

3In this thesis, 1KiB = 2^10 bytes = 1,024 bytes; 1MiB = 2^20 bytes = 1,048,576 bytes; 1GiB = 2^30 bytes = 1,073,741,824 bytes. [IEC Technical Committee (TC) 25, 2000]
2.5 Compression in Practice

while distances are encoded with another set. For each block, DEFLATE chooses between static or semi-static Huffman codes. If semi-static Huffman codes are selected, then the prelude is situated at the beginning of the compressed block. A publicly available library called ZLIB [Gailly and Adler, 1995] is available, which allows DEFLATE to be compiled into any program. The default compression effectiveness of GZIP can be improved by adding the –9 option. The –9 option forces GZIP to search more exhaustively for a longer match. All experiments with GZIP in this thesis use version 1.3.2 with the –9 option.

The PPMD system by Alistair Moffat (University of Melbourne) couples a PPM mechanism based on escape method D with an arithmetic coder. Two parameters can be set by the user to alter the program’s behaviour. The first parameter specifies the maximum context, while the second one sets a limit on the amount of memory that can be used. Context nodes are created by PPMD until the memory limit is reached. If this happens while a message is being processed, then all context nodes are deleted, and compression continues with a fresh trie. Experiments in this investigation make use of a seventh order model with a memory limit of 255 MiB, both chosen to favour compression effectiveness.

The BZIP2 program [Seward, 1996] combines block sorting with a Huffman coder. The block sorting mechanism divides the message into blocks ranging from 100,000 to 900,000 bytes in size, depending on the value of a command line argument. In this thesis, the program parameter –9 is always given so that a block size of 900,000 bytes is used. The documentation that accompanies the program states that the memory required by BZIP2 for encoding with this block size is around 7.4 MiB, while decoding requires about 3.6 MiB. A compression library similar to ZLIB called LIBBZIP2 is available from the BZIP2 web site. The version of BZIP2 used in this thesis is 1.0.2. Details of the implementation of BZIP2 are given by Seward [2000] and Seward [2001].

Two additional programs are purely entropy coders which presume a zero-order model, and hence that all modelling processes have already been allowed for. Input to these coders is a sequence of integers. The SHUFF system [Moffat and Turpin, 1997, Turpin and Moffat, 2000] is a semi-static Huffman coder that compresses an integer sequence in blocks in two passes. The end of a block is determined either by an explicit size from the user, or when
2.6 Experimental Framework

Experiments are performed in this thesis in order to validate the compression and retrieval ideas presented. All tests are carried out on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache running Debian Linux. Programs implemented for this thesis are written in the C programming language and compiled with version 2.95.4 of the GNU gcc compiler with the -O3 optimisation flag. Compression systems have been evaluated in terms of efficiency and effectiveness. Unless otherwise stated, compression times are expressed in CPU seconds, averaged over three trials. CPU seconds reflects the amount of time the computer spent on the program, but excluding the time spent accessing files, for example. Due to the small variations in execution times, an average calculated from three runs was sufficient for most of the experiments.

Compression ratios throughout the investigation are primarily expressed in one of two different ways. If the original message is composed of 32-bit integers, then the ratios are expressed in bits per symbol (bps). However, for most of this thesis, the message is encoded in Latin-1 and the units employed are bits per character (bpc).

The test data that was chosen for this investigation included the Large Canterbury Corpus\textsuperscript{5} (LCC) and data from the Text REtrieval Conference\textsuperscript{6} (TREC). The Large Canterbury Corpus includes three files: \texttt{E.coli}, \texttt{world192.txt}, and \texttt{bible.txt}. The \texttt{E.coli} file consists of the complete genome of the \textit{E.coli} bacterium and only contains the four

\textsuperscript{4}One publicly available PPM system which employs escape method D is at \url{http://www.cs.waikato.ac.nz/~singlis/}
\textsuperscript{5}URL: \url{http://corpus.canterbury.ac.nz/}
\textsuperscript{6}URL: \url{http://trec.nist.gov/}
2.6 Experimental Framework

characters a, c, g, and t, with no whitespace. The file world192.txt is the CIA world fact
book, while bible.txt is the King James version of the bible. The sizes of these three
files are listed in Table 2.5.

Three data files were drawn from the news articles of disks 1 and 2 of TREC's TIPSTER
collection [Harman, 1995]. The largest data file is NEWS, which is composed of news
articles from the Wall Street Journal (WSJ) from 1987 to 1992, as well as articles from the
Associated Press (AP) from 1988 and 1989. All of the news articles have been marked up
with the Standard Generalized Markup Language (SGML) [International Organization
for Standardization, 1986]. Due to the wide range in years covered by the articles, and
the fact that the SGML tags differ between WSJ and AP, there are some discontinuities
of style caused by the concatenation. However, the homogeneous data collection of news
articles permits a data file of about 1,000 MiB in size to be processed. Small excerpts
from WSJ and AP are shown in Figure 2.7. The remaining two data files from TREC are
WSJ508 and WSJ20. The WSJ508 file includes all of the Wall Street Journal articles from
TREC, which is 508 MiB in total. A smaller data file, WSJ20, consists of just the first
20 MiB of WSJ508.

The three compression systems, GZIP, BZIP2, and PPMD, were applied to these 6 data
files. The results are presented in Table 2.5 and Table 2.6. Since SHUFF and UINT are
coding mechanisms and not general-purpose compression systems, experiments with them
are detailed later in Chapter 3 with different data files.

Table 2.5 shows the exact file sizes of the 6 files and the compression effectiveness
attained with the compression programs. Both GZIP and BZIP2 were executed with the
-9 option, while PPMD used a seventh order model and 255 MiB of memory in order to
favour compression effectiveness. For all 6 files, the statistical model of PPMD consistently
performed the best, followed by BZIP2, and then GZIP. This order is expected, given their
respective models and coders, as well as the amount of memory available for them, and
the consequential use of very large blocks by PPMD.

Table 2.6 examines the compression experiments with respect to time in CPU seconds,
averaged over three trials on the test machine discussed earlier. In almost every scenario,
Text Compression

Figure 2.7: Two excerpts from the News data set.

GZIP required the least amount of time for both encoding and decoding, followed by BZIP2, and then PPMD. The notable exception is the compression of E.coli, where GZIP required the most time for encoding. This phenomenon is due to the small set of characters in the file, which made looking for the longest match more time consuming.

The tables demonstrate a common trade-off in compression between efficiency and effectiveness. Generally, the best compression effectiveness comes at the cost of execution
2.7 Effect of Compression on Browsing

<table>
<thead>
<tr>
<th>Filename</th>
<th>File size</th>
<th>GZIP</th>
<th>BZIP2</th>
<th>PPMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.coli</td>
<td>4,638,690</td>
<td>2.240</td>
<td>2.158</td>
<td>2.034</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2,473,400</td>
<td>2.333</td>
<td>1.584</td>
<td>1.454</td>
</tr>
<tr>
<td>bible.txt</td>
<td>4,047,392</td>
<td>2.326</td>
<td>1.671</td>
<td>1.564</td>
</tr>
<tr>
<td>WSJ20</td>
<td>20,971,520</td>
<td>2.908</td>
<td>2.078</td>
<td>1.656</td>
</tr>
<tr>
<td>WSJ508</td>
<td>533,196,049</td>
<td>2.941</td>
<td>2.105</td>
<td>1.637</td>
</tr>
<tr>
<td>NEWS</td>
<td>1,048,491,001</td>
<td>2.963</td>
<td>2.145</td>
<td>1.674</td>
</tr>
</tbody>
</table>

Table 2.5: Compression ratios for the Large Canterbury Corpus, WSJ20, WSJ508, and NEWS using GZIP, BZIP2, and PPM, expressed in bits per character. File sizes are indicated in bytes.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Encoding (s)</th>
<th>Decoding (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GZIP</td>
<td>BZIP2</td>
</tr>
<tr>
<td>E.coli</td>
<td>14.0</td>
<td>4.4</td>
</tr>
<tr>
<td>world192.txt</td>
<td>0.6</td>
<td>2.1</td>
</tr>
<tr>
<td>bible.txt</td>
<td>1.8</td>
<td>3.4</td>
</tr>
<tr>
<td>WSJ20</td>
<td>6.4</td>
<td>17.9</td>
</tr>
<tr>
<td>WSJ508</td>
<td>170.0</td>
<td>447.3</td>
</tr>
<tr>
<td>NEWS</td>
<td>323.3</td>
<td>871.5</td>
</tr>
</tbody>
</table>

Table 2.6: Compression times for the Large Canterbury Corpus, WSJ20, WSJ508, and NEWS using GZIP, BZIP2, and PPM. Times are reported in CPU seconds on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache, averaged over three trials.

time. These results also indicate why these three programs were selected for this chapter. Besides demonstrating the three types of models mentioned earlier, the GZIP system was chosen for its speed, while PPMD was chosen for its compression effectiveness. The BZIP2 system demonstrates the compromise in time and space between the other two systems. Throughout this thesis, these two tables are frequently referred to as benchmarks for these compression systems and the test data.

Finally, note that GZIP is particularly fast for decoding, a trait that is common with most dictionary-based algorithms, and one that forms the foundation for much of this thesis, as explained in the next section of this chapter.

2.7 Effect of Compression on Browsing

The central theme in this thesis is the exploration of the compromises required in order to accommodate both compression and browsing. The compression systems surveyed in
this chapter can reduce a message to less than a fifth of its original size. However, several aspects of their underlying algorithms preclude any form of retrieval, let alone browsing, without performing a complete decompression.

The support for searching a compressed message is important, since time and space could be saved while still being able to obtain information from the compressed representation. The solution offered by the zgrep script [Free Software Foundation, 2001], is to decompress a reduced message created by gzip temporarily, and then search this copy with a standard pattern matcher. However, while such a solution is convenient for the user, the solution is somewhat unsettling given that the compressed message is fully decoded. The source of the problem is the compression algorithms themselves.

The main drawback to the dictionary-based and statistical models presented is their adaptive, on-line nature. In order to retrieve from an arbitrary point in a message processed with either LZ77, LZ78, or PPM, everything before it must be decoded. In the case of block sorting, only the block where the point of interest is located must be decoded. So, extending this solution to the other algorithms, creating more blocks would improve retrieval feasibility. Generalising the solution, synchronisation points could be inserted throughout the compressed message so that retrieval from a location in the compressed stream only requires a full decode from the last synchronisation point. However, since synchronisation points indicate a location in the compressed message where the model is re-built, as the frequency of these points increase, compression can be expected to suffer.

Furthermore, while compression systems are measured in terms of effectiveness and efficiency, retrieval systems are usually interactive. That is, users are more concerned with response times and expect a short waiting period for results to appear. The importance of access time provides another reason why a full decode of the compressed document is discouraged.

In addition to these two points, coding algorithms present a third problem, which is byte-alignment. Even though the channel alphabet is based on the binary system, computers access data in bytes consisting of 8 bits each. The low-level operations on bits which are intrinsic to static and entropy coding algorithms also affect both retrieval and
2.7 Effect of Compression on Browsing

decoding times.

These three problems are considered in the remainder of this thesis, beginning with Chapter 3. At the heart of this investigation is the RE-PAIR algorithm of Larsson and Moffat [2000], which is described next. The algorithm is dictionary-based and has many similarities with LZ77 and LZ78. However, RE-PAIR possesses other qualities that make it ideal for browsing. The chapters that follow Chapter 3 build on the algorithm and address the three problems posed here.
Chapter 3

Compression by Recursive Pairing

RE-PAIR is a dictionary-based compression algorithm developed by Larsson and Moffat [2000]. RE-PAIR operates off-line and commences the compression process only after the entire message has been seen. This is in contrast to dictionary-based algorithms such as LZ77 and LZ78, described in Chapter 2, which build dictionaries on-line.

Off-line compression algorithms have gained attention recently for two reasons. First, the amount of memory available in computers has steadily increased in recent years. Earlier on-line algorithms were able to operate with just enough memory to perform the compression, but not enough to also hold the entire message. With more memory to use, off-line algorithms which examine an entire message before starting compression offer an alternative to on-line algorithms. Second, decompression occurs more frequently than compression in retrieval systems. In these circumstances, an underlying off-line compression mechanism that operates slower than its on-line counterparts but ensures fast decompression may be preferred. As with on-line dictionary-based algorithms, off-line ones build a dictionary of phrases. Occurrences of these phrases in the message are replaced by references to them. However, off-line dictionary-based algorithms are faster because the model does not have to be updated throughout decompression.

RE-PAIR transmits the dictionary explicitly as a separate stream to the receiver instead of incrementally like LZ77. This results in two outputs: the dictionary, and a sequence of references to entries in the dictionary. These two parts are coded independently and transmitted to the decompressor, DES-PAIR, as two distinct streams in order to reconstruct.
Compression by Recursive Pairing

the message. As is discussed later in this chapter, it is RE-PAIR’s separation of the message into two separate streams that makes it ideal for phrase browsing.

The remainder of the chapter is structured as follows. Section 3.1 provides an overview of the RE-PAIR algorithm. While recent computers have more memory available for off-line algorithms to use, memory is still limited and Section 3.2 gives a detailed analysis of the data structures used by RE-PAIR. Those data structures provide a balance between memory space and execution time. Section 3.3 describes how the two outputs of RE-PAIR are represented as coded sequences. Section 3.4 gives experimental results for the implementation of RE-PAIR used, and Section 3.5 describes other algorithms similar to RE-PAIR. Finally, Section 3.6 discusses the features of RE-PAIR that make it attractive for fast searching and retrieval, and the issues that prevent it from being used for that purpose.

3.1 The Re-Pair Model

RE-PAIR reduces the length of a message by replacing repeated pairs of symbols with a new symbol which does not appear in the message. Initially, the most frequently occurring pair of characters is located and all occurrences of it are replaced with a new symbol. Each replacement reduces the length of the message by one. The new symbol is added to a dictionary, together with a record of how it should be expanded. This process is repeated, and the next most frequently occurring pair of symbols (now including any new symbols introduced by RE-PAIR) is replaced. The process terminates when no pair of adjacent symbols occurs more than once in the message. The source of the name “RE-PAIR” arises from the recursive pairing nature of the algorithm.

In order to describe RE-PAIR in greater detail, some more precise definitions and notations are necessary. The alphabet ($\Sigma$) of the original message are also called primitives. RE-PAIR creates and adds phrases to the dictionary. The set of all phrases is denoted as $\rho$.

Primitives and phrases are collectively called symbols, and are denoted using, for example, $\alpha$, $\beta$, $\delta$, and $\gamma$. If it is important to indicate that a symbol $\alpha$ is a phrase
introduced by RE-PAIR and not a primitive, then it is circled, @. If the order in which
the phrase was added to the dictionary is important, then the phrase is instead indicated
by a circled number, †.

Every symbol $\alpha$ has a generation $g(\alpha)$ which indicates how deep it is in the hierarchy
of paired symbols. By definition, the generation of any primitive is zero. The generation
of a phrase $\alpha$ is one more than the higher generation of its two components.

The outputs of RE-PAIR are a dictionary, and a sequence of references to entries in
the dictionary. The dictionary contains all of the primitives and phrases composed out
of them by the pairing process. Since every phrase consists of two symbols from a lower
generation (either shorter phrases or primitives), the dictionary resembles a hierarchy of
phrases. To underscore this fact, the dictionary generated by RE-PAIR is called the phrase
hierarchy, denoted as $P$. The list of pointers to symbols in the phrase hierarchy is called
the reduced sequence, and is denoted as $S$.

A sample application of RE-PAIR is shown in Table 3.1. The sample message used is
part of the “Woodchuck” message of Figure 2.2. The initial message is shown in the first
entry of the first column. Each subsequent row of the table indicates the identification of
the most frequently occurring pair of symbols in the previous row, and the replacement
of all occurrences of it with a new symbol. The new phrase added to the phrase hierarchy
as a result of that set of replacements is shown in the second column. For each phrase,
the frequency with which it occurs in the message is shown in the third column, and
the generation of that phrase is shown in the fourth column. As expected, the sequence
of replacements performed by RE-PAIR is in decreasing frequency. In this example, the
generation numbers of the phrases are in increasing order, but this is not always the case.

Several phrases from the same generation may occur with equal frequencies. The order
in which they are chosen depend on the data structures used by RE-PAIR. Discussion of
the method for breaking ties appears in Section 3.2.

After the last replacement in the table, no pair of symbols occurs more than once. RE-
PAIR then terminates, and the final sequence (last entry in the first column) is encoded.
The phrase hierarchy is also a necessary part of the coded message, which includes all of
Compression by Recursive Pairing

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Phrase hierarchy</th>
<th>( f )</th>
<th>( g(\alpha) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>how much wood could a woodchuck chuck</td>
<td>1 \to c h</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>how wood could a woodchuck chuck</td>
<td>2 \to d w</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>how wood could a woodchuck chuck</td>
<td>3 \to w</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>how wood could a woodchuck chuck</td>
<td>4 \to c k</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>how wood could a woodchuck chuck</td>
<td>5 \to o o</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>how wood could a woodchuck chuck</td>
<td>6 \to u</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>how wood could a woodchuck</td>
<td>7 \to \alpha</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>how wood could a woodchuck</td>
<td>8 \to \beta</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1: A sample application of Re-Pair on part of the “Woodchuck” message. Each line represents the replacement of all occurrences of a pair of symbols. The effect on the sequence and the phrase hierarchy is shown, along with the frequency of the symbol pair, and the generation number of the replacing phrase.

the second column as well as a description of the set of primitives.

The Decompressor

The Re-Pair decompressor (nicknamed, for good reason, Des-Pair) is simpler than the compressor. After decoding the stream representing the phrase hierarchy, the entire structure is built within memory so that each phrase requires two words of memory to indicate its two components.

A symbol in the reduced sequence is decoded by expanding the phrase hierarchy until the string of primitives that it represents are found. A recursive traversal for each symbol in the reduced message can be slow, depending on the size of the phrase hierarchy and the generation of each symbol. A more economical approach is to decode every phrase in the phrase hierarchy first, so that a decoding table could be made. Then, for each symbol in the sequence, each decoded symbol can be copied directly from the decoding table to the output stream. This method is significantly faster, and is similar to the LZ78 algorithm discussed in the previous chapter. A third approach, which strikes a balance between the other two, maintains a buffer of recently decoded symbols for later use. When a symbol is decoded, the buffer is consulted first before it is recursively expanded.

Table 3.2 presents some experimental results which compare the differences in time
and memory usage of these three methods. The test data is WSJ20, which was compressed in experiments described later in Section 3.4. The compressed message has 90 primitives, 310,486 phrases, and 1,904,118 references. The longest phrase was composed of 1,257 primitives, and there were 21 generations in the phrase hierarchy. The third method listed in Table 3.2 employs a 256 KiB buffer of recently expanded symbols. Decompression time was measured in CPU seconds, averaged over 3 trials. Memory is stated in MiB, and represents the maximum memory consumed during execution.

Expanding the phrase hierarchy before decoding the sequence was only slightly faster than employing a buffer of recently expanded symbols. The difference in time is expected to be larger if the reduced sequence had more symbols. That is, the cost of fully expanding symbols is recovered when more symbols are decoded. Recursively expanding the phrase hierarchy with each symbol requires more time, but less memory. The difference in memory between the first and the third methods is equal to the size of the buffer. Also, note that the amount of extra memory required by the second method is equal to the size of the document due to the recursive nature of the phrase hierarchy.

The results show that the creation of a modest buffer offers a compromise between execution time and memory usage. This approach has been adopted in the implementation of DES-PAIR for later experiments.

3.2 Analysis of Re-Pair

As with any compression system, implementation choices in RE-PAIR allow a range of trade-offs between memory space and processing time. The three different approaches to the implementation of DES-PAIR is one example of the choices available. A similar
Compression by Recursive Pairing

balance is required by RE-PAIR between memory space and execution time. This section discusses data structures described by Larsson and Moffat [2000], and the trade-offs that they allow.

Data Structures

The three main data structures used by RE-PAIR are shown in Figure 3.1, with part of the text of Table 3.1 used as an example. The purpose of the data structures is to locate active pairs and active phrases efficiently. An active pair is a pair of symbols in the sequence that is under consideration for replacement. Likewise, an active phrase is a potential replacement for all active pairs which have the same left and right components. For example, if the pair αβ appears 10 times in the sequence, each of the occurrences is an active pair, and the 10 active pairs are associated with a single active phrase. Active phrases are required when considering a set of replacements, while active pairs are required for a particular replacement.
3.2 Analysis of Re-Pair

The first data structure is an array of sequence nodes, shown along the top of Figure 3.1. Initially, each sequence node holds a primitive from the message so that for a message of \( n \) primitives, \( n \) sequence nodes are required. During compression, a pair of adjacent symbols is replaced with a single symbol. Each sequence node contains a symbol and two pointers. The pointers serve two purposes. Active pairs are formed by taking pairs of adjacent sequence nodes so that every node (except for the first and last ones) can be a left component and a right component of an active pair. The first purpose of the sequence node pointers is to form a circular double-linked list of active pairs so that each node points to the left components of the previous and next equivalent active pairs. In Figure 3.1, the first “ch” active pair shown is the second of three that appear in the “Woodchuck” message. When the active phrase “ch” is chosen, each occurrence of “ch” can be replaced in \( O(1) \) time. As active pairs are replaced, gaps appear in the sequence. The second purpose of the node pointers is to allow each gap to be traversed in \( O(1) \) time, as illustrated in Figure 3.2. Four snapshots of the sub-sequence of Figure 3.1 are shown at various points during the recursive pairing process. As gaps appear in the sequence array, the pointers at either end of each gap lead to the other end of the gap. The nodes within a gap are unused, and are coloured grey.

At the centre of Figure 3.1 are active phrase nodes. An active phrase node contains a pointer to its active pairs, as well as a count of the number of active pairs. Each active phrase needs to be locatable either by its active pair count or its components. To accomplish this, each node is placed on two circular double-linked lists. One list permits fast look-up of symbol pairs through a hash table, with collisions resolved by chaining. The other list is placed in a priority queue data structure implemented as an array. Each slot in the priority queue corresponds to active phrases of a given number of active pairs. The number of slots in the priority queue is equal to the number of active pairs associated with the active phrase with the most active pairs. The active phrases in a list at a given priority queue slot are placed so that the oldest active phrase node is at the head of the list. Active phrase nodes are attached to the hash table and priority queue using circular double-linked lists, so as to allow \( O(1) \) time addition and removal from their respective
lists, once they have been located.

A fourth data structure, not shown in the figure, records the active phrases that have been added to the phrase hierarchy. Each phrase in the phrase hierarchy contains pointers to its two component phrases and its generation number.

**Space Analysis**

The amount of space used by the four main data structures of **Re-Pair** is linear in the length of the message. To see this, suppose that a message of \( n \) primitives is to be processed. The greatest number of active phrases throughout the processing of the message is \( p \). The size of the hash table is fixed at \( h \) slots. The most active pairs an active phrase can have is \( \lfloor n/2 \rfloor \), which is equal to the maximum number of slots for the priority queue. The most active pairs occur when every primitive in the message is the same. In this case, there are \( n - 1 \) pairs of primitives, but only half of them are made active because **Re-Pair** prevents the existence of overlapping pairs. For example, when there are three consecutive symbols with the same value, only one active pair exists. Overlapping pairs
3.2 Analysis of Re-Pair

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Space used</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>$3n$</td>
</tr>
<tr>
<td>active phrases</td>
<td>$6p$</td>
</tr>
<tr>
<td>hash table</td>
<td>$h$</td>
</tr>
<tr>
<td>priority queue</td>
<td>$\lfloor n/2 \rfloor$</td>
</tr>
</tbody>
</table>

Table 3.3: The amount of memory used by the most important data structures in Re-Pair, in words. The initial message has $n$ symbols. The most active phrases in memory during compression is $p$. The hash table has $h$ slots.

are elaborated further below.

The amount of memory used by the main data structures and the active phrase nodes in words is summarised in Table 3.3. Since an active phrase node is larger than a phrase in the phrase hierarchy, the space occupied by an active phrase node can be re-used by its corresponding phrase in the phrase hierarchy. So, the memory required for the phrase hierarchy is not shown in Table 3.3. Initially, as many as $n - 1$ active phrases may be created, each with one active pair. However, as none of them would be candidates for replacement, Re-Pair would then terminate. Indeed, in order to minimise memory usage, Larsson and Moffat [2000] showed how only useful active phrases that have at least two active pairs each would be created by performing an additional pass through the message.

While the exact relationship between $p$ and $n$ cannot be determined theoretically, empirically, it can be shown that $n$ is the dominant term for average text. To investigate the relationship, Re-Pair was used to process WSJ20. (Details of these experiments appear in Section 3.4.) For WSJ20, $|\Sigma| = 90$, $|\rho| = 310,486$, and $n = 20,971,520$. Figure 3.3 shows the number of active phrases in memory after a certain number of phrases have been added to the phrase hierarchy by Re-Pair for WSJ20. The number of phrases that have been added to the phrase hierarchy (horizontal axis) was sampled at regular intervals. The total number of active phrases peaks at 482,370 after 13,137 phrases have been created in the hierarchy. Afterwards, the number of active phrases decreases until the last replacement, which creates symbol number 310,486, deletes the last active phrase. The size of the priority queue is initialised to the most frequent symbol, which for WSJ20, occurred 466,082 times.
Compression by Recursive Pairing

![Graph showing the relationship between the number of active phrases and the number of phrases added to the phrase hierarchy for WSJ20.](image)

Figure 3.3: The relationship between the number of active phrases and the number of phrases added to the phrase hierarchy for WSJ20.

Using the above values for $n$ and $h$, if $p = 482,370$, then $n$ is easily the most important factor with respect to memory usage. The sequence nodes occupy the most space out of all the data structures. Therefore, RE-PAIR requires $3.5n + O(p)$ words of memory.

The memory space analysis presented is slightly different from the one given by Larsson and Moffat [2000] due to minor changes to the data structures used. For example, Larsson and Moffat limited the size of the priority queue to $\sqrt{n}$ words of memory so that the last list in the priority queue contains active phrase nodes which have $\sqrt{n}$ or more active pairs. This final list is again ordered so that the least recently created active phrase node is at the head of the list. However, to locate the next active phrase on the list with the most active pairs would require a linear search through the list. Instead, a larger priority queue was chosen to avoid this linear search. Larsson and Moffat also showed a method of periodically compacting the sequence nodes if memory becomes a serious constraint. By reclaiming the space occupied by the gaps, no new active phrase nodes need to be created after the replacement process has begun. Each compaction operation requires an additional pass over the sequence node array to locate the gaps and an update of the pointers of each sequence node.

A shorter priority queue and sequence node compaction reduces the amount of memory used by RE-PAIR by using additional execution time. Neither of these approaches were
incorporated into the implementation of RE-PAIR for this thesis because a higher priority was given to execution time. Furthermore, if longer messages are processed which require more memory than available, then an alternative method is employed, which is described in Chapter 5.

**Time Analysis**

The data structures described in the last section allow a message to be processed efficiently. With the data structures described, the RE-PAIR algorithm is shown in Algorithm 3.1.

An analysis of the time complexity of the algorithm is as follows. In the first step, an initial pass over the sequence is performed which considers every symbol as a left and a right component with its two immediately adjacent symbols. Active phrases are created if necessary and inserted into the hash table. Then, in step 2, a pass is performed over the hash table to insert active phrases into the priority queue lists.

The dominant step of the algorithm is the recursive pairing loop that begins at step 3. Each replacement reduces the length of the sequence by one symbol. During a single replacement, the most costly steps involve adding, removing, and updating active phrases. By using double-linked lists in the hash table and priority queue, all of these operations execute in $O(1)$ expected time. Therefore, RE-PAIR runs in time linear to the length of the message.

Furthermore, the number of active phrases created by replacing the $k$ active pairs of $\alpha\beta$ can never exceed $k$. That is, after the priority queue has been initialised in step 2, it does not enlarge in size. Also, once an active phrase has been placed on the priority queue list at slot $k$, it is guaranteed to be used by RE-PAIR for pairing or moved to a lower slot in the priority queue. That is, RE-PAIR is guaranteed to terminate.

The algorithm also stipulates how active phrases with equal frequency counts are handled. A “least recently used” policy is enforced by adding active phrases to the end of the priority queue lists. This ensures that active phrases are processed first in decreasing frequency, and then in the order in which they were encountered by RE-PAIR. An exception to this rule are generation 1 active phrases. Generation 1 active phrases are created in
Compression by Recursive Pairing

Algorithm 3.1: The RE-PAIR algorithm.

1. Scan the message and create up to \( n - 1 \) active pairs. Create active phrases as necessary and add them to the hash table.

2. Add every hashed active phrase with at least two active pairs to the appropriate position in the priority queue. Active phrases that have less than the minimum number of active pairs are deleted.

3. Recursively pair symbols as follows until no pair of adjacent symbols occur twice or more in the sequence.
   
a) Retrieve the first active phrase node from the list located at the last priority queue slot that is in use. This node corresponds to the active phrase, \( \alpha \beta \), which currently has the most active pairs. Let the replacement be \( \ominus \).
   
i. Locate the next active pair for \( \alpha \beta \) and its context, \( \delta \alpha \beta \delta \).
   
ii. Decrement the active phrase frequency counts of \( \delta \alpha \) and \( \beta \delta \) and move them one priority queue list to the left. Remove any active phrase with a frequency count of 0.
   
iii. Replace \( \alpha \beta \) with \( \ominus \).
   
iv. Increment the active phrase frequency counts of \( \delta \ominus \) and \( \ominus \delta \) and insert them to the end of the appropriate priority queue lists. Create new active phrases if necessary.
   
b) Add the phrase \( \ominus \rightarrow \alpha \beta \) to the phrase hierarchy.
   
c) Remove the active phrase \( \alpha \beta \) from the hash table and priority queue.
   
d) Remove all active phrases with counts less than 2.

4. Transmit the phrase hierarchy and the reduced sequence using an appropriate representation.

step 1 but are added to the priority queue in step 2 from the first slot in the hash table to the last.

Problems with Re-Pair

There are two problems with the RE-PAIR algorithm which were not discussed by Larsson and Moffat [2000]. They arise in certain isolated cases. They appear because of RE-PAIR's handling of overlapping pairs. Overlapping pairs occur when a sequence of length 3 or
more of a single symbol appear in the message. During the initial scanning, the second of two overlapping pairs is not considered as an active pair. For example, in the string “a_1a_2a_3” (where the subscripts are used to distinguish between occurrences of “a”), “a_1a_2” is the only active pair. A false count would result if these three primitives were considered as two active pairs.

However, suppose a section of a sequence appears like this: \( \delta \alpha \beta_1 \beta_2 \beta_3 \delta \). Every pair of symbols is an active pair except for the underlined pair because of the three consecutive \( \beta \) symbols. If the replacement \( \odot \rightarrow \alpha \beta \) is made next, then the sequence becomes \( \delta \odot \beta_2 \beta_3 \delta \). According to the algorithm, after the replacement, two new active pairs are considered: \( \delta \odot \) and \( \odot \beta_2 \). But with \( \beta_1 \) replaced, \( \beta_2 \beta_3 \) should be considered an active pair. While the algorithm looks for new active pairs by examining the immediately preceding and following symbols to a replacement, when overlapping pairs are concerned, this example shows that a look-ahead of two symbols from where the replacement was made is required to ensure that an active pair is not missed.

Unfortunately, longer sequences of consecutive symbols may require more look-ahead. Suppose there are five consecutive symbols as in the following section of a sequence: \( \delta \alpha \beta_1 \beta_2 \beta_3 \beta_4 \beta_5 \delta \). Two pairs of symbols are inactive, as shown. As before, suppose \( \alpha \beta \) is replaced, resulting in: \( \delta \odot \beta_2 \beta_3 \beta_4 \beta_5 \delta \). The consecutive sequence of four \( \beta \)’s now only contains one active pair. After the replacement, the sequence should have been examined up to four symbols after \( \odot \) so that the active pair \( \beta_3 \beta_4 \) gets replaced by the two active pairs \( \beta_2 \beta_3 \) and \( \beta_4 \beta_5 \). The amount of look-ahead required, though, depends on the number of consecutive symbols. In the implementation of Re-Pair used, a look-ahead of just two symbols is performed, and when more consecutive symbols appear in a message, active pairs are missed. However, a missed active pair causes a problem only if it would have been replaced. Suppose in the above example, \( \beta_5 \delta \) is replaced by \( \odot \). The sequence then becomes \( \delta \odot \beta_2 \beta_3 \beta_4 \odot \). Three \( \beta \)’s remain with only one active pair among them, which is correct. Even though the active pair \( \beta_4 \beta_5 \) was missed, it was unimportant because of a replacement involving \( \beta_5 \).

The problem of consecutive symbols causing missed active pairs that should have been
Compression by Recursive Pairing

replaced generally occurs for only whitespace in average text. In the 20 MiB of WSJ20 mentioned earlier, only one active pair was missed which could have been replaced. The problem may be more serious for pathological cases and binary data, such as graphic files. The word-based parsing for Re-Pair described in the next chapter make this situation even more rare, though still possible.

The second problem with the Re-Pair algorithm is the appearance of duplicate phrases caused by consecutive symbols. After the replacement of $\alpha \gamma$ with $\ominus$, three consecutive symbols may appear multiple times in the sequence, requiring another replacement. However, based on what pairs of consecutive symbols are active and which are inactive, two phrases may result. The first replacement might be $\ominus \rightarrow \alpha \gamma_1$ and the second might be $\ominus \rightarrow \gamma_1 \alpha$. These two phrases are identical if they are expanded to primitives. This problem is also not handled by the implementation of Re-Pair used in this thesis, but a resolution is discussed in Chapter 5.

3.3 Coding for Re-Pair

The phrase hierarchy and the sequence produced by the Re-Pair model need to be coded separately before transmission. The coding requirements of each are different because of how they are used. The phrase hierarchy needs to be fully decoded due to the interdependency between symbols across generations. On the other hand, it is possible that only small sections of the sequence are required during retrieval. Any part of the sequence can be displayed as long as the phrase hierarchy is already available. This decoding requirement for the sequence is set aside for the moment and is considered in Chapter 6.

The Phrase Hierarchy

Larsson and Moffat [2000] discuss a variety of methods for encoding the phrase hierarchy. The best method, chiastic slide followed by interpolative coding, is used in the implementation of Re-Pair considered in this thesis.

Interpolative coding [Moffat and Stuiver, 2000] encodes a sorted list of non-uniformly
distributed integers by first encoding the median, and then recursively representing the
two half lists of values which are less than or greater than the median. Since the number
of possible values is restricted by the endpoints of the intervals, the medians within each
interval can be encoded within the range established by its endpoints instead of the entire
list. This allows shorter codes to be used when large number of values cluster together.

The two components of each phrase in the phrase hierarchy form a two-dimensional
space of approximately $(|\Sigma| + |\rho|)^2$ in size. To apply interpolative coding each phrase is
mapped to a single number using its left $(l)$ and right $(r)$ components. The method used
is called the chiastic slide, denoted as $\chi(l, r)$.

Chiastic slide values are calculated incrementally starting from the first generation of
phrases. The way that generations are defined ensures that each phrase has at least one
of its components in the immediately preceding generation, as otherwise it would have
been placed in an earlier generation. To calculate the chiastic slide values for phrases in
generation $i$, the two-dimensional space where at least one component is in generation
$i - 1$ is enumerated. The set of primitives is also interpolative coded, but does not need
to be mapped using the chiastic slide. Instead, they are encoded using their underlying
values. For example, the primitives of the text message of Table 3.1 are encoded as a
sorted list of their Latin-1 representations.

Figure 3.4 shows how chiastic slide is used to enumerate the primitives and the first
generation of phrases for the example of Table 3.1. On the left is a list of all possible left
components and along the bottom is a list of all possible right components.

The set of all possible first generation phrases (unshaded region) can be calculated once
all of the primitives have been sorted and encoded. The five boxed numbers in this region
represent the five phrases from the first generation. This sorted list $[0, 13, 45, 46, 114]$ is
then interpolative coded.

The encoding of the first generation of phrases is a special case since only one previous
generation (the primitives) exists. The encoding of subsequent generations all resembles
the encoding of the second generation (shaded region). Phrases in the second generation
must have either both components in the first generation (dark grey region) or one com-
<table>
<thead>
<tr>
<th>left component ($l$)</th>
<th>w 10</th>
<th>u 9</th>
<th>o 8</th>
<th>m 7</th>
<th>l 6</th>
<th>k 5</th>
<th>h 4</th>
<th>d 3</th>
<th>c 2</th>
<th>a 1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>oo</td>
<td>9 19</td>
<td>29</td>
<td>39</td>
<td>49</td>
<td>59</td>
<td>69</td>
<td>79</td>
<td>89</td>
<td>99</td>
<td>109</td>
<td>118</td>
<td>125</td>
</tr>
<tr>
<td>ch</td>
<td>8 18</td>
<td>28</td>
<td>38</td>
<td>48</td>
<td>58</td>
<td>68</td>
<td>78</td>
<td>88</td>
<td>98</td>
<td>108</td>
<td>117</td>
<td>124</td>
</tr>
<tr>
<td>ck</td>
<td>7 17</td>
<td>27</td>
<td>37</td>
<td>47</td>
<td>57</td>
<td>67</td>
<td>77</td>
<td>87</td>
<td>97</td>
<td>107</td>
<td>116</td>
<td>123</td>
</tr>
<tr>
<td>d_</td>
<td>6 16</td>
<td>26</td>
<td>36</td>
<td>46</td>
<td>56</td>
<td>66</td>
<td>76</td>
<td>86</td>
<td>96</td>
<td>106</td>
<td>115</td>
<td>122</td>
</tr>
<tr>
<td>s_w</td>
<td>5 15</td>
<td>25</td>
<td>35</td>
<td>45</td>
<td>55</td>
<td>65</td>
<td>75</td>
<td>85</td>
<td>95</td>
<td>105</td>
<td>114</td>
<td>113</td>
</tr>
</tbody>
</table>

**Figure 3.4:** Example of chiastic slide for the example of Table 3.1. The chiastic slide values for the first two generations of phrases are shown in boxes.
ponent in an earlier generation (light grey regions). Since it is more likely that one of the
two components of a phrase is from a different generation, the chiastic slide assigns values
starting from where one of the components is zero up to the top right corner of the dark
grey region. The sorted list [98,110] is used to represent the second generation of phrases;
they are assigned symbol numbers 16 and 17.

In order to assign chiastic slide values for the current generation, \( i \), the size of the
shaded region is required. This region is bounded by the total number of primitives and
phrases up to and including the immediately preceding generation, \( g_{i-1} \), and up to but
excluding it, \( g_{i-2} \). The size of the shaded region is thus \( g_{i-1}^2 - g_{i-2}^2 \). In the example, to
encode the second generation of phrases, \( i = 2 \), \( g_1 = 16 \), \( g_0 = 11 \), and \( g_2^2 - g_0^2 = 134 \).

The formula used to calculate the chiastic slide, adapted from Larsson and Moffat, is
shown in Equation 3.1. The first two cases are the light grey regions, while the last two are
used for the dark grey region. While calculating the chiastic slide requires multiplication
and squaring operations (and division and square roots for decoding), many of the terms
are calculated per generation and not per phrase.

\[
\chi(l,r) = \begin{cases} 
2l(g_{i-1} - g_{i-2}) + g_{i-1} - 1 - r & \text{for } l < g_{i-2} \\
(2r + 1)(g_{i-1} - g_{i-2}) + l - g_{i-2} & \text{for } r < g_{i-2} \\
(l(2g_{i-1} - l)) + g_{i-1} - 1 - r - g_{i-2}^2 & \text{for } g_{i-2} \leq l \leq r \\
(r(2g_{i-1} - r - 2)) + g_{i-1} - 1 + l - g_{i-2}^2 & \text{for } g_{i-2} \leq r < l
\end{cases}
\] (3.1)

In order to use interpolative coding with the chiastic slide, the phrase hierarchy needs
to be sorted twice. First, the phrases are sorted by generation number, since they were
added to the phrase hierarchy by decreasing frequency and not increasing generation.
Second, for each generation, the phrases are assigned values using the chiastic slide, and
are sorted by these values before being interpolative coded. As shown in Figure 3.4, after
the primitives and phrases have been assigned chiastic slide values and sorted by them,
they are enumerated starting from 0 (shown along the two axes). A final pass through the

61
Compression by Recursive Pairing

sequence is also required to ensure it uses the re-numbered phrase hierarchy, rather than the one that corresponds to decreasing pair frequency.

The encoding of the phrase hierarchy resembles a context-free grammar in Chomsky normal form (CNF) [Chomsky, 1959]. In CNF, every production rule consists of a non-terminal which generates either two other non-terminal, or a single terminal. The encoding of phrase “ch” represents $3 \rightarrow 2 \cdot 4$ while the primitive “c” is represented as $2 \rightarrow \text{"c"}$. It is for this reason that each entry in the phrase hierarchy can be considered to be a rule. The significant differences between a RE-PAIR phrase hierarchy and a more general CNF is that the phrase hierarchy is used to generate a single message using the sequence as a starting rule whereas a context-free grammar in CNF is usually expected to generate a language with more than one string in it.

The Reduced Sequence

The other output of the RE-PAIR process is the reduced sequence. As the primitives and phrases are enumerated from 0 at the end of the chiastic slide encoding, references in the reduced sequence are just integers which refer to a rule in the phrase hierarchy. Since the purpose of RE-PAIR is to remove redundancy in the message, no pair of symbols appears more than once in the reduced sequence. An entropy coder using a zero-order model on the symbols in the sequence captures the great majority of the remaining structure. Certainly, using a higher order model would be counter productive. The semi-static Huffman coder and the arithmetic coder described in the last chapter can be used without further modification to the sequence.

3.4 Experiments

Experiments were conducted on the Large Canterbury Corpus and WSJ20 on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache machine, described in the last chapter. Larger test data could not be compressed because of the amount of memory used by RE-PAIR. The next chapter demonstrates how this limitation can be overcome by
3.4 Experiments

| Filename    | $|\Sigma|$ | $|\rho|$ | $|S|$ | Number of generations | Longest phrase (in primitives) |
|-------------|---------|---------|-------|------|-----------------------|-------------------------------|
| E.coli      | 4       | 67,368  | 652,665 | 20  | 1,800                 |
| world192.txt| 94      | 55,473  | 212,648 | 19  | 432                   |
| bible.txt   | 63      | 81,229  | 386,095 | 19  | 548                   |
| WSJ20       | 90      | 310,486 | 1,904,118 | 21 | 1,257                 |

Table 3.4: Statistics from experiments with Re-PAIR.

| < DOC>•<DOCNO>•WSJ 870 324 20001</DOCNO>•<HL>•<DD>•03/2
John.Blair Is Near Accord To Sell Unit, Sources Say
</HL>•<DD>•<SO>•WALL STREET JOURNAL(J)</SO>•<IN>•
REL •TENDER OFFERS, MERGERS, ACQUISITIONS (TNM) •<-
MARKETING, ADVERTISING (MKT) •TELECOMMUNICATIONS, •<-
BROADCASTING, TELEPHONE, TELEGRAPH (TEL)
</IN>•<DATELINE>•NEW YORK</DATELINE>•<TEXT>•...
John.Blair &n; Co. is close to an agreement to_|

Figure 3.5: Expanding the first 20 symbols of the WSJ20 sequence into primitives. For the purposes of display, • indicate linefeeds and long phrases are broken at the locations indicated by ↔.

partitioning the message into blocks. For all experiments described in this chapter, each message is processed as a single block.

Some statistics from applying Re-PAIR on these files are shown in Table 3.4. As an example of the phrases formed by Re-PAIR, the beginning of WSJ20 is presented in Figure 3.5.

The compression ratios achieved by Re-PAIR on these four files are shown in Table 3.5. The last three columns of this table represent compression ratios for BZIP2, GZIP, and PPMD, taken from Table 2.5 on page 41. Recall that GZIP and BZIP2 were executed with the -9 option, while a seventh order model for PPMD was used to ensure the best possible compression. Execution of PPMD was limited to 255 MiB of memory, so as to effectively remove the restriction on its memory usage and reduce the chance of requiring the model to be re-constructed. For the results reported in Table 3.5, the model was not re-built by PPMD. While Re-PAIR has the potential to use more memory than any of these compression systems, they are reasonable benchmarks due to their widespread popularity.
Compression by Recursive Pairing

| Filename   | RE-PAIR | | | | | |
|------------|---------|---------|---------|---------|---------|
|            | Phrase hierarchy | Total using SHUFF | Total using UINT | GZIP | BZIP2 | PPMd |
| E.coli     | 0.108 | 2.086 | 2.081 | 2.240 | 2.158 | 2.034 |
| world192.txt | 0.312 | 1.624 | 1.597 | 2.333 | 1.584 | 1.454 |
| bible.txt  | 0.290 | 1.763 | 1.749 | 2.326 | 1.671 | 1.564 |
| WSJ20      | 0.223 | 1.774 | 1.770 | 2.907 | 2.078 | 1.656 |

Table 3.5: Compression ratios for the Large Canterbury Corpus and WSJ20 using RE-PAIR. Total compression ratios for RE-PAIR include the phrase hierarchy and the coded sequence with either SHUFF or UINT. Compression results from the last three columns have been taken from Table 2.5.

| Filename   | Encoding | | | | | |
|------------|----------|---------|---------|---------|---------|
|            | RE-PAIR | SHUFF | UINT | RE-PAIR | SHUFF | UINT |
| E.coli     | 16.2 | 0.3 | 1.1 | 0.4 | 0.1 | 1.2 |
| world192.txt | 9.1 | 0.1 | 0.4 | 0.3 | <0.1 | 0.4 |
| bible.txt  | 15.0 | 0.3 | 0.9 | 0.4 | 0.1 | 1.0 |
| WSJ20      | 88.5 | 1.6 | 6.7 | 2.5 | 0.4 | 7.5 |

Table 3.6: Compression and decompression times for the Large Canterbury Corpus and WSJ20 using RE-PAIR and DES-PAIR in seconds. Times have been averaged over three trials.

Compression effectiveness using RE-PAIR is better than GZIP in all cases and varies with BZIP2, depending on the file. This is an expected result given the differences in the amount of memory used. PPMd achieves better compression than RE-PAIR, but as mentioned in the last chapter, statistical models can be expected to outperform dictionary-based ones. The arithmetic coder gives better compression than the Huffman coder. Note also that the phrase hierarchy is small compared to the original message and the reduced sequence, occupying less than 20% of the final message for all of the files.

Compression and decompression times for RE-PAIR are shown in Table 3.6 in CPU seconds, averaged over three trials. Even though the coding of the phrase hierarchy is logically separate from the RE-PAIR process, in the implementation of RE-PAIR used, the coding of the phrase hierarchy is tightly coupled with RE-PAIR, and so, the execution times of RE-PAIR and DES-PAIR both include the encoding and decoding times of the phrase hierarchy.

Comparing the compression times of RE-PAIR with the other programs (Table 2.6), RE-PAIR takes more time even if entropy coding is not taken into account. However,
decoding with the two dictionary-based systems, DES-PAIR and GZIP, are fast. Of the two, GZIP is consistently faster. Furthermore, the encoding and decoding times are higher for arithmetic coding compared to Huffman coding. Because of this speed difference and the very slight difference in compression, the entropy coder used by experiments in later chapters is the Huffman coder. The downside of this decision is the slightly worse compression, as shown in Table 3.5.

3.5 Related Work

Other than operating off-line, several other differences exist between RE-PAIR and the dictionary-based algorithms of Chapter 2. First, RE-PAIR explicitly separates the dictionary from the sequence of references, producing two streams that together, make up the compressed message. Second, every primitive and phrase must be used at least once in order to appear in the RE-PAIR phrase hierarchy. In contrast, phrases may be added to the dictionaries formed by LZ77 or LZ78, but never referred to.

Storer and Szymanski [1982] showed that, in a dictionary-based compression algorithm, determining the phrases that would achieve the best compression for a given message is NP-hard. Therefore, like LZ77 and LZ78, many dictionary-based compression algorithms use heuristics for phrase construction. Bentley and McIlroy [2001] considered a pre-processor to compression which identifies long repeated strings appearing far apart.

Wolff [1975] described a program called MK10 for identifying breaks in text in the absence of any whitespaces. The system operates similar to RE-PAIR by identifying and recursively replacing pairs of symbols. However, unlike RE-PAIR, the message is scanned from left to right and replacements are made after a pre-determined threshold has been met (for example, 10 occurrences). Later, Wolff [1977] extended this work by selecting the most frequent pair of symbols after one complete scan of the message. However, since the primary goal of Wolff was the segmentation of text, little attention was given to the compression effectiveness or efficiency of MK10.

The incremental encoding of Rubin [1976], like RE-PAIR, uses frequency of digrams (pairs of symbols) and operates through recursive pairing for compression. However, more
Compression by Recursive Pairing

emphasis is placed on conditions for program termination rather than a compact method of storing the dictionary. Some of these conditions include phrase frequency and phrase length. There were also few details given on an efficient implementation with respect to time or memory space. However, as an indication of the amount of time and space that such an algorithm would require, Rubin [1976] noted that the methods used have "fairly high computation time and large storage requirements".

Manber [1997] used digrams for searching in compressed text. Even though Latin-1 is a character set of size 256, half of it is unused by most English documents. The remaining 128 values were assigned to frequently occurring pairs of primitives and used to compress a message. The search pattern was similarly compressed so that a general-purpose string matching algorithm could be used to locate occurrences of the pattern within the compressed message. The problem of overlapping pairs is avoided by ensuring that a symbol which is a left component of one pair of symbols cannot be a right component of another pair. Earlier, Gage [1994] devised a technique called Byte-Pair Encoding (BPE) which resembles the algorithm by Manber, except that searching was not considered.

More recently, the Off-Line mechanism of Apostolico and Lonardi [2000] showed how an annotated trie for a message could be used to locate candidates for a dictionary. The phrases in the dictionary were created directly from substrings of the message without referring to other phrases in the dictionary. Central to Off-Line is the computation of a gain measure. The gain measure determines the next best phrase to add to the dictionary using criteria such as substring length, substring frequency, and pointer costs to all occurrences. Three methods of dictionary encoding were employed, one of which stored the phrases as literal text in an external dictionary. Turpin and Smyth [2002] extended this work by looking at other methods of computing the gain measure.

The Ray system [Cannane and Williams, 2001] also compresses a message off-line, but through multiple passes so that the message itself does not need to be stored in memory. It builds a dictionary by recursively replacing diagrams, similar to Re-Pair. Each pass over the message is roughly equivalent to a generation in Re-Pair. The Ray algorithm consists of three steps: accumulation of frequencies, the identification of candidates, and
replacement of symbol pairs. The first step is only performed once since the sequence's statistics are updated as replacements are made. The remaining two steps are repeated until either no pair of symbols appears twice or more, or a preset limit on generations has been reached. Pairs of symbols of equal frequency were handled differently than REPAIR. If the three symbols $\alpha\beta\delta$ appear in the sequence and step 2 is being performed, $\alpha\beta$ is identified as a candidate for replacement only if $\alpha$ occurs with an equal or higher frequency than $\beta$. If this is not the case, then this pair of symbols is skipped, and the next pair, $\beta\delta$, is considered. The RAY dictionary is encoded using Elias and Huffman codes.

Finally, operating on-line instead of off-line is the SEQUITUR algorithm of Nevill-Manning and Witten [1997]. The SEQUITUR algorithm builds a dictionary resembling a grammar, but unlike REPAIR, a rule can generate more than two symbols. During compression, as each message symbol is read, the grammar is updated according to two rules: digram uniqueness and rule utility. Digram uniqueness ensures that no pair of adjacent symbols appear more than once in the grammar. Digram uniqueness effectively adds phrases to the dictionary as symbols appear in the sequence (rule $S$). The creation of rules can cascade from rule $S$, and results in a hierarchical grammar. Rule utility prohibits any rule from being used only once. When a rule created earlier ends up appearing once in the right-hand sides of all of the rules, it is expanded at the location it is used and then removed from the grammar. Adherence to rule utility results in the right-hand sides of rules becoming longer than just digrams. Rule $S$ is transmitted to the receiver while the rest of the grammar is sent implicitly using back-pointers, similar to a LZ77 implementation. Also, Nevill-Manning and Witten [2000] compared on-line and off-line techniques for SEQUITUR by using phrase heuristics such as most frequent, longest, and most compressive.

An example of SEQUITUR is shown in Table 3.7 using the same “Woodchuck” message as the one from the example of Table 3.1. Each row of the table indicates the current rule $S$, the SEQUITUR rule that was violated, and the action required to fix the grammar. The first row shows the first time a rule violation has occurred. If a rule already exists when digram uniqueness is enforced, no new rule is added to the dictionary, as shown with the
Compression by Recursive Pairing

<table>
<thead>
<tr>
<th>Rule applied to grammar</th>
<th>Action applied to grammar</th>
<th>Rule enforced</th>
<th>Rule</th>
<th>digram uniqueness</th>
</tr>
</thead>
<tbody>
<tr>
<td>how.much.wood.could.</td>
<td>Add 1 → d.</td>
<td>digram uniqueness</td>
<td>how.much.woo(1coul(1a.w</td>
<td>Add 2 → w</td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.wo</td>
<td>Add 3 → o</td>
<td>how.much.woo(1coul(1a.wo</td>
<td>Expand 2: 3 → wo</td>
<td></td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.5</td>
<td>Add 4 → 5o</td>
<td>how.much.woo(1coul(1a.5</td>
<td>Replace rule 2</td>
<td></td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.5ch</td>
<td>Add 6 → ch</td>
<td>how.much.woo(1coul(1a.5ch</td>
<td>Use rule 6</td>
<td></td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Add 7 → 6c</td>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Expand 6: 7 → 6uck</td>
<td></td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Add 8 → 7k</td>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Remove rule 6</td>
<td></td>
</tr>
<tr>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Add 9 → 8</td>
<td>how.much.woo(1coul(1a.5ch6uck.6ch</td>
<td>Remove rule 7</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: A sample application of the Sequitur algorithm on the message used in Table 3.1. Message symbols are read one at a time and appended to rule 5, shown in the first column. Each row represents the moment when one of the two rules of Sequitur (digit uniqueness or rule utility) need to be enforced. The Sequitur rule that is violated and the action performed to enforce it are shown in columns 2 and 3, respectively.

third occurrence of “ch”. After phrase 5 is added to the dictionary, phrase 2 is used only once. Phrase 2 is expanded in phrase 3 and removed from the dictionary because of rule utility. At the end of this example, the rules sent to the receiver are 5, 3, 4, 5, 6, 7, and 8.

While the Sequitur algorithm differs from R-PAIR in the above points, a system for phrase browsing has also been developed called Phind [Nevill-Manning et al., 1997]. The methods used by Phind, discussed later in Chapter 7, are similar to the ones for R-PAIR, but differences emerge due to their underlying compression mechanisms.

3.6 Retrieval with Re-Pair

In Chapter 2, a wide range of algorithms for modelling were presented. This chapter adds another dictionary-based model to the list, which is more suited for both retrieval and browsing.
3.6 Retrieval with Re-Pair

The most important characteristic of the Re-Pair algorithm is that recursive pairing based on frequency is more appropriately implemented in a semi-static manner, so that the entire message is available. As a consequence of this, the compressed message is composed of two separate streams, which each has its own advantages. The phrase hierarchy is small and occupies only 3% of the space taken by the original message. Moreover, the phrase hierarchy contains a significant amount of information about the document. The hierarchical structure of the phrase hierarchy means that it describes the many relationships symbols form with lower and higher generation symbols. The phrase hierarchy is not only crucial for decompression, but appears suitable for phrase browsing, as well. The second stream is the reduced message of references to the dictionary.

Searching the reduced sequence has been examined by Manber [1997] and Shibata et al. [1999], who based their work on algorithms similar to Re-Pair. While Manber modified the pattern prior to searching, Shibata et al. presented two approaches. First, they expanded all corresponding encodings of the pattern before applying a pattern matching algorithm. Second, they used the Knuth-Morris-Pratt automaton [Knuth et al., 1977] for the pattern to aid the search. In addition to searching, the reduced sequence can be decoded from any reference, provided the phrase hierarchy has been processed already.

At the conclusion of the last chapter, three requirements were mentioned which a compression algorithm should satisfy in order to be adapted for retrieval. First, the algorithm should allow decoding from an arbitrary point in the compressed message, which is satisfied by the structure of the two streams from Re-Pair. In contrast, since algorithms like PPM and block sorting rely on symbol contexts, a random point in the compressed representation cannot be decoded without examining everything before. Alternatively, a compressed message made with block sorting can be pre-processed in-memory so that an indexing structure is available for searching [Ferragina and Manzini, 2000]. The LZ77 and LZ78 algorithms are both dictionary-based like Re-Pair, but they were designed to be adaptive and so, combine the dictionary and the references into a single stream.

The second requirement is that decoding should be fast so that the amount of time that a user of a retrieval system waits is minimised. While Re-Pair is slower than GZIP for
decoding, a comparison between the results of Table 3.6 and Table 2.6 show that RE-PAIR is noticeably faster than BZIP2 and PPMD.

Finally, the alignment of codewords to bits rather than bytes would facilitate more efficient decoding. Experiments in this chapter compressed the reduced sequence with SHUFF and UINT. Even though Huffman coding was advocated after examining the decoding times of SHUFF and UINT, an alternative coding mechanism which is able to decode the sequence faster may be possible which sacrifices some compression effectiveness. As for the phrase hierarchy, the good compression effectiveness attained with interpolative coding make the lack of any byte-alignment acceptable.

There are, however, four changes to RE-PAIR before it can be included into a retrieval system. First, the use of frequency alone to select phrases for the phrase hierarchy presents problems for retrieval. In a retrieval system, it is expected that the query terms match the primitives or phrases in the phrase hierarchy. The solution to this problem requires the attention to shift to how RE-PAIR can be forced to select words and phrases which resemble the ones displayed to the user in the interface.

The second change deals with the problem of memory space required by RE-PAIR. While RE-PAIR operates in space linear in the length of the message, the constant factor is quite high. Assuming four bytes per word, compressing WSJ20 requires 240 MiB for the sequence nodes alone. In order for a retrieval system to be useful, files one or even two orders of magnitude larger may need to be processed using reasonable amounts of memory.

The third improvement is with the entropy coding applied to the sequence. While phrase browsing is performed solely with the phrase hierarchy, searching for those phrases involves the reduced sequence. By using an entropy coder, if appearances of a phrase are being sought, the reduced sequence needs to be sequentially decoded. Instead, a coding mechanism is required which allows efficient searching through the reduced sequence.

Finally, an interface is required which allows users to browse phrases and then locate and examine the contexts in which they occur.

Solutions to each of these four problems are investigated separately in the next four
3.6 Retrieval with Re-Pair

chapters, in the order they have been listed. Chapter 4 describes alternatives to RE-PAIR's method of selecting phrases by first aligning phrases to whitespaces, and then by using a word-based pre-processing scheme using punctuation as a guide. Chapter 5 shows how a message can be partitioned into blocks by RE-PAIR so that larger messages can be compressed. Compressed blocks are merged in order to produce a single phrase hierarchy for browsing. Chapter 6 examines alternatives to entropy coding the entire reduced message to allow fast searching for phrases. And finally, Chapter 7 describes phrase browsing and an interface which ties the above components together.
Chapter 4

Selecting Phrases for Browsing

The previous chapter showed how RE-PAIR reduces a message by repeatedly selecting active phrases, replacing their corresponding active pairs with new symbols, and adding these new symbols to the phrase hierarchy. Active phrases are chosen in order of decreasing number of active pairs, with ties broken by choosing the active phrase created first.

RE-PAIR's approach to phrase construction uses primitives as the smallest building block in the phrase hierarchy. In the previous chapter, primitives were defined as the Latin-1 representations used in the message. However, people do not view messages as characters, but as clusters of alphabetic characters which form words with whitespace characters in between. A user that browses phrases drawn from a RE-PAIR phrase hierarchy should be presented with meaningful words rather than arbitrary strings of characters that might not align with word boundaries. In order to accomplish this, RE-PAIR needs to treat whitespace and non-whitespace characters differently.

This chapter considers the problem of how RE-PAIR can build a phrase hierarchy made up of useful phrases for browsing. The policy used by RE-PAIR to select phrases based on decreasing frequency alone is insufficient for this purpose. Three alternative methods are considered which complement RE-PAIR's phrase selection heuristic.

The remainder of this chapter is structured as follows. In Section 4.1, an overview of information retrieval and compression systems which operate on words rather than characters is provided, along with the definition of "word" that is used in this chapter. Then, the three methods of extending RE-PAIR are examined. In Section 4.2, the first
Selecting Phrases for Browsing

method adds rules to RE-PAIR in order to align words to the word boundaries implied by the non-words that surround them. The second method introduces a word-based pre-processing phase which explicitly separates a document into two streams: one for words and the other for non-words. In Section 4.4, a third method combines the first two methods by using the word-based pre-processing step with a new set of rules for punctuation marks. Section 4.5 compares these three methods, and shows the impact of the changes through experiments. Finally, the chapter concludes with a summary of the three methods described.

4.1 Forming Words from Characters

RE-PAIR, as described in the last chapter, views documents at the character level. However, people read and understand documents as a sequence of words. In a retrieval system, the query terms that a user provides and the results that the system returns should be in the form of words. Just as index-based retrieval systems build indexes using words, so should a phrase browsing system using RE-PAIR.

Information retrieval systems generally parse their input into three different classes of symbols: word symbols, inter-word symbols, and special processing symbols [Kowalski, 1997]. Documents are treated as an alternating sequence of word symbols and inter-word symbols (or non-word symbols). Some retrieval systems also make use of special processing symbols for separating documents [Witten et al., 1999] or for indicating meta-data, such as SGML tags.

Symbols are encoded individually by a compression system which assume a memoryless source. Because the characters that form words in a document do not appear independently, word-based mechanisms have also been employed for compression. A word-based compression system requires choices to be made with respect to how the document is parsed. When a document is parsed as words, similar to a retrieval system, suitable definitions of words, non-words, and special processing symbols (if any) are required. These different symbol types may exist in the reduced message as different streams, or as a single, unified stream. Finally, some methods of parsing a message into words replaces
the words with references to a lexicon (or dictionary) that also needs to be encoded and transmitted.

Bentley et al. [1986] devised a word-based parsing scheme which separated a message into two alternating streams: one of alphanumeric symbols and another of non-alphanumeric symbols. A move-to-front scheme (MTF) followed by Huffman coding was used to encode word references. Moffat [1989] undertook experiments on the Bentley et al. scheme, and also extended character-based PPM to word-based PPM and found improvement in compression effectiveness with a first-order model, but little additional improvement with a second-order model. An alternating sequence of words and non-words was also used, with a 20 character limit on each. Hospool and Cormack [1992] investigated word-based compression for LZ78, where the lexicon was transmitted implicitly by using an escape mechanism to indicate the appearance of a new word. They also investigated first-order context modelling by conditioning a word based on the parts-of-speech (noun, verb, article, adjective, or other) of the preceding word. The parts-of-speech of words were determined through dictionary look-up. Words that did not appear in the dictionary were assigned to the category that would achieve the best compression. Isal and Moffat [2001] devised a compression scheme that combined block sorting with a word-based model. Words and non-words were limited to 12 characters each and a single lexicon was produced by adapting the spaceless words approach of de Moura et al. [2000]. The spaceless words approach combines the word and non-word lexicons into a single lexicon. During decoding, a space character is added after every word. If a word is followed by an explicitly coded non-word, then no space is added and the non-word is decoded in its place instead. Isal and Moffat [2001] reported a compression ratio of 1.69 bpc for WSJ20 when combining word-based block sorting with a suitable coder.

Nevill-Manning and Witten [1997], parsed semi-structured documents into words so that SEQUITUR could be applied to references to words in the word lexicon. In semi-structured data, words may be conditioned on other words that are not immediately adjacent. As a trivial example, a closing tag in an SGML message appears because of an opening tag earlier in the message. SEQUITUR used heuristics based on the amount
Selecting Phrases for Browsing

of gain that could be obtained from a distant word prediction. Experiments showed that
SEQUITUR attained better compression effectiveness than other systems, including GZIP
and a PPM implementation using escape method C, for a sample semi-structured document.
Compression effectiveness further improved when the heuristics were employed.

The changes to RE-PAIR described in this chapter reflect previous work in word-based
compression. However, some different choices were made because the goal of this inves-
tigation is to balance phrase browsing with compression. Before discussing word-based
compression for RE-PAIR, some definitions are required. A word character is any alphabetic
or numeric character in the Latin-1 character set. In order to accommodate the
SGML test data used in this thesis, the three characters “<”, “>”, and “/” are also con-
sidered to be word characters. Any other symbol, including whitespace and punctuation
marks, are non-word characters. No special processing symbols are defined at this time.
A sequence of contiguous word (or non-word) characters is a token. A document is viewed
as an alternating sequence of word tokens and non-word tokens.

The original version of RE-PAIR described by Larsson and Moffat [2000] and in Chap-
ter 3 is called character-based RE-PAIR. This distinction is required in order to differen-
tiate it from three additional word-aware versions of RE-PAIR that are described in the
next three sections.

4.2 Word-aligned Re-Pair

The first method of processing a document as words is called word-aligned RE-PAIR.
Word-aligned RE-PAIR augments rules to the part of the RE-PAIR algorithm that decides
which pairs of symbols are permitted as active pairs.

Figure 4.1 provides the motivation behind this variation of RE-PAIR. The figure
expands the phrases generated by character-based RE-PAIR in Table 3.1 on page 48 into
all primitives. The phrases in Figure 4.1 are listed in the order in which they are added
to the phrase hierarchy. Three of these eight phrases contain the space character, a non-
word symbol. While browsing phrases with non-word symbols is visually unappealing, a
more noticeable problem is its effect on the choice of phrases made by RE-PAIR. Since
the phrases “wo” and “od” in Figure 4.1 occur with a higher frequency in the message than “wo” and “od”, the likelihood of the word “wood” from being formed as a phrase is reduced. The problem is more obvious with longer documents and is caused by all non-word symbols, not just the space character.

Word-aligned Re-Pair solves this problem by introducing the four rules shown in Table 4.1. As each phrase is added to the phrase hierarchy, it is labelled as a word symbol (W), a non-word symbol (N), or a mixture (M). The primitives (characters) are also labelled as “W” or “N” during initialisation. When two adjacent symbols are being considered as potential active pairs, the word-aligned rules are consulted. The second and third columns represent the classification of the two symbols under consideration. The first and fourth columns represent the surrounding context in which that pair of symbols occurs in the sequence. An “X” indicates that the classification is unimportant. If the pair of symbols satisfies any one of the four rules, then it can become an active pair. The fifth column of the table indicates the category of the corresponding symbol after it has been added to the phrase hierarchy. For example, when an active phrase representing active pairs composed of two non-word symbols is added to the phrase hierarchy, it is also classified as a non-word symbol. The last column provides examples which satisfy each rule, with boxes representing the two symbols that are being considered as an active pair.

The alignment of words is caused by the fourth rule. The fourth rule ensures that a word symbol can only be combined with a non-word symbol, forming a mixed symbol, if both of them are complete. A complete word is a sequence of one or more word symbols,
Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th>Before-left</th>
<th>Left</th>
<th>Right</th>
<th>After-right</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>N</td>
<td>N</td>
<td>X</td>
<td>N</td>
<td>woodchuck</td>
</tr>
<tr>
<td>X</td>
<td>W</td>
<td>W</td>
<td>X</td>
<td>W</td>
<td>woodchuck</td>
</tr>
<tr>
<td>X</td>
<td>M</td>
<td>M</td>
<td>X</td>
<td>M</td>
<td>woodchuck</td>
</tr>
<tr>
<td>N</td>
<td>M</td>
<td>W</td>
<td>N</td>
<td>W</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 4.1: The rules used by word-aligned RE-PAIR for deciding when a pair of symbols can become active pairs. If a pair of adjacent symbols satisfy any of the four rules, then it becomes an active pair.

such that on both sides of it, there are only non-words. A similar definition applies to non-words. As there is only one such rule, words can only be left components of mixed symbols, and non-words can only be right components. The first three rules simply allow longer phrases of the same type to be formed.

These four rules are applied at two locations in the RE-PAIR algorithm of Algorithm 3.1. They are used when the message is initially scanned (step 1), and after an active pair replacement is made and the frequency counts of the newly introduced symbol with its adjacent neighbours is incremented (step 3(a)iv). This ensures that the active pair counts for active phrases only include active pairs that satisfy one of these rules. During recursive pairing, even though the sequence changes from the time when an active pair is identified to when it is replaced, the classification of active pairs remains correct. The first three rules classifies pairs of symbols regardless of their contexts. More importantly, changes in the sequence do not affect pairs classified using the fourth rule. Suppose a pair of symbols is classified as a mixed pair when the symbol preceding the left component is a non-word symbol. Later, when this pair is replaced, the preceding non-word symbol may have become part of a mixed symbol. But since all mixed phrases start with a word symbol and end with a non-word symbol, the fourth rule still holds for this active pair.

Table 4.2 shows word-aligned RE-PAIR applied to the same “Woodchuck” message, with the third column showing the expanded phrases. The expansion of phrase @ is the word “wood”, as desired. The pair “d...” does not exist as a phrase because, in this message, “d” does not appear by itself as a complete word. As another example, Figure 4.2 shows the expanded phrases generated by word-aligned RE-PAIR when it is applied to
4.2 Word-aligned Re-Pair

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Phrase hierarchy</th>
<th>Expanded phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>how much wood could a woodchuck</td>
<td></td>
<td></td>
</tr>
<tr>
<td>how much could a woodchuck</td>
<td>1 → ch</td>
<td></td>
</tr>
<tr>
<td>how much could a wood chuck</td>
<td>2 → od</td>
<td></td>
</tr>
<tr>
<td>how much could a wooduck</td>
<td>3 → ck</td>
<td></td>
</tr>
<tr>
<td>how much could a wooduck</td>
<td>4 → wo</td>
<td></td>
</tr>
<tr>
<td>how much could a wooduck</td>
<td>5 → chu</td>
<td></td>
</tr>
<tr>
<td>how much could a wood</td>
<td>6 → wood</td>
<td></td>
</tr>
<tr>
<td>how much could a wood</td>
<td>7 → chuck</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Applying word-aligned Re-PAIR to the same brief “Woodchuck” message used in Table 3.1 on page 48.

the complete “Woodchuck” message (see page 18). The word-alignment rules ensure that word symbols and non-word symbols continue to grow until they are surrounded by symbols of a different type. Then, mixed symbols are created by having complete word symbols combine with complete non-word symbols on the right. In the figure, every symbol represents phrases that are either part of an English word, a complete English word, or an alternating sequence of complete words followed by complete non-words. In this example, the longest non-word symbol is only one symbol long (the space character, <), so complete non-word phrases that are longer than one primitive do not exist in the phrase hierarchy.

In a phrase hierarchy with |Σ| primitives and |ρ| phrases, word-aligned Re-PAIR requires an extra (|Σ| + |ρ|) words of memory compared to character-based Re-PAIR. Every primitive and phrase in the phrase hierarchy is assigned a category. Since the categories can be represented in 2 bits, an implementation which conserves memory is possible which only uses an extra 2(|Σ| + |ρ|) bits. Categories could also be assigned to each symbol in the sequence. However, for a message of length n, this approach requires n words of memory, which is typically larger than (|Σ| + |ρ|). Furthermore, augmenting categories to the sequence is unnecessary because the category of a symbol in the sequence is the same as its category in the phrase. The phrase “wood” is a word symbol (W), regardless of the context in which it appears. When an occurrence of “wood” is considered as a component of an active pair, then its adjacent neighbours in the sequence are checked.
Figure 4.2: Expanded phrases generated by word-aligned Re-PAIR on the complete “Woodchuck” message, in the order in which they were added to the phrase hierarchy. In front of each phrase, its word-aligned category is shown. No phrases in this example are non-words (N).

There are two advantages to word-aligned Re-PAIR. First, the alignment performed by word-aligned Re-PAIR operates transparent to the decompressor. The augmented rules only affect the compressor. Second, word-aligned Re-PAIR allows compound words like “woodchuck” to be split up into its constituent words, provided they exist elsewhere in the message by themselves.

Experimental results with word-aligned Re-PAIR are provided at the final section of this thesis. However, problems with word-aligned Re-PAIR exist, and are addressed in the next section by word-based Re-PAIR.

### 4.3 Word-based Re-Pair

Word-aligned Re-PAIR creates phrases that are more useful than those of character-based Re-PAIR, by aligning phrases to word boundaries. However, several problems remain in the phrase hierarchy which require a different solution. First, two words used in the complete “Woodchuck” message cannot be found in the phrase hierarchy of Figure 4.2: “how” and “would”. Each word exists in the message only once and while “how” can be
4.3 Word-based Re-Pair

located by performing a direct search on the reduced sequence using primitives, “would”
cannot be found since it has been broken into two phrases: ② and ③. Second, punctuation
was removed in the complete “Woodchuck” message, but if it was included, then the
phrase “wood,” is treated differently from “wood?”. To a user, both phrases have the
same meaning, except the second one marks the end of a sentence. Furthermore, case
information of letters was removed from the message, but word-aligned Re-Pair would
have treated “Wood,“ as being a different phrase from “wood,“.

That is, while word-aligned Re-Pair is an improvement over character-based Re-
Pair, some problems remain. The essence of the problems is that word-aligned Re-Pair
does not force a message into words, but merely applies a set of rules only when a decision
is required. With respect to the first problem above, since “how” and “would” each
appears once in the message, no decision was required by Re-Pair, resulting in neither
word appearing in the phrase hierarchy. Instead, a word-based scheme is required, similar
to the ones used by other compression systems described earlier. Word-based compression
requires an additional processing stage where a message is divided into word tokens and
non-word tokens. The processing can occur as the message is compressed, or it may
complete before compression begins.

The pre-processing used by word-based Re-Pair needs to satisfy two criteria in or-
der to strike a balance between the requirements of retrieval and compression. First, the
phrases produced by Re-Pair after the pre-processing should improve compared to ones
created by the character-based and word-aligned Re-Pair variants. For example, phrases
should be aligned on word boundaries. Second, like other compression algorithms, com-
pression must be lossless. Figure 4.3 shows a structure that satisfies these two conditions.

In Figure 4.3, processes are shown as rectangles, while the flow of data are indicated
as lines. Each process is further elaborated later in this section. Initially, the message is
separated into two streams of tokens: one for words and the other for non-words. The
word tokens are then case folded and stemmed. Lexicons for words and non-words are
maintained separately and updated as the message is processed. The word references are
then compressed by Re-Pair. The punctuation flags, shown as dashed lines, are explained
in Section 4.4. All of these steps are performed by the pre-processor, dubbed PRE-PAIR, to emphasise its use before RE-PAIR.
4.3 Word-based Re-Pair

Separating the Message

The message is parsed as an alternating sequence of word and non-word tokens. A token is at most 16 characters in length. For longer words (and non-words), a special zero-length token exists in both types of tokens. For example, if a word of length 21 characters is encountered, then it is split so that two words of lengths 16 and 5 are encoded, with a zero-length non-word in between. This parsing scheme is similar to the one described by Moffat [1989].

Furthermore, to accommodate the SGML test files used by this thesis, the definition of words is extended. The "<" character specifies the start of a new word token while the ">" character specifies the end of one, regardless of the context in which they occur. This definition ensures that SGML tags are tokens, regardless of whether or not there is whitespace surrounding them.

Case folding

After a word token has been identified, it is case folded to lower case. Case folding is a technique commonly used by retrieval systems for constructing and accessing an index. It permits a user to enter query terms without considering whether the case of the words being given to the system matches the case of the indexed terms. However, in retrieval systems, the index is separate from the message that it indexes. So, the case folding can be a lossy transformation because it need not be reversed. In contrast, the case folding done by PRE-PAIR needs to be reversible since RE-PAIR is used for constructing phrases for both browsing and compression. Therefore, a modifier is produced for each word token when case folded. When the modifier is applied to the case folded token, the original word is produced.

Since any character in a word might be in upper case and so require folding, a modifier of at least 16 bits is required, with each bit corresponding to a character position. A bit value of "1" means that the corresponding source character is in upper case. The bit flags are in reversed order with respect to the characters in the word token. That is, a word token with only the first character in upper case results in a case folding modifier with
Selecting Phrases for Browsing

the least significant bit turned on. This ensures that all word tokens with only the initial character in upper case have the same case folding modifier, regardless of word length. Furthermore, an extra bit is used to indicate the special case when every character is in upper case, so that the case folding modifier is 17 bits in length. While encoding a token, every character is checked. When decoding, as soon as no other bits in the modifier are set to “1”, the rest of the word is not processed.

Reversible case folding is also used by the Length Index Preserving Transform (LIPT) of Awan and Mukherjee [2001]. LIPT is a compression pre-processor that uses a static dictionary of lower case words. As most words that contain upper case characters either have only the first character in upper case or all of the characters in upper case, special symbols are inserted into the message to indicate this. For other combination, individual characters are flagged if they are in upper case. PRE-PAIR operates similar to LIPT except that the case folding modifiers of PRE-PAIR exist as a stream separate from the words. Also, PRE-PAIR assigns a bit flag to every character that is processed while LIPT adds a character modifier only if an upper case character is found.

Stemming

After the word token is case folded, it is stemmed. Stemming, or conflation [Frakes, 1992] is a technique used in information retrieval to remove word suffixes to produce its root form. For example, “searches”, “searching”, and “search” all stem to the word “search”. This allows users to query a retrieval system without worrying about entering the correct ending for a word. Moreover, stemming also reduces the number of entries in the lexicon.

Several authors have proposed stemming algorithms [Lovins, 1968, Dawson, 1974, Porter, 1980]. Of these, the Porter stemming algorithm is used in this chapter, based upon the recommendation of Kowalski [1997].

All stemming algorithms conflate a word by applying steps which successively remove endings. A step is applied if the word matches a set of conditions. Some conditions include the length of the word, the characters in the suffix, or the characters before the suffix. The algorithms do not consider the semantic meaning of the word, and each
4.3 Word-based Re-Pair

<table>
<thead>
<tr>
<th>Original word</th>
<th>Root word obtained</th>
<th>Expected root word</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>charact</td>
<td>character</td>
</tr>
<tr>
<td>frequency</td>
<td>frequenc</td>
<td>frequent</td>
</tr>
<tr>
<td>retrieval</td>
<td>retriev</td>
<td>retrieve</td>
</tr>
<tr>
<td>searchable</td>
<td>searchabl</td>
<td>search</td>
</tr>
<tr>
<td>skies</td>
<td>ski</td>
<td>sky</td>
</tr>
<tr>
<td>pies</td>
<td>pi</td>
<td>pie</td>
</tr>
</tbody>
</table>

Table 4.3: Some unexpected results from the Porter stemming algorithm. Words are listed in the first column. The root form obtained by the algorithm is shown in the second column, while the third column displays the root form a person most likely would provide.

word in a document is stemmed independently of adjacent words. This is acceptable for most information retrieval systems since the stemmed word is used internally to search through an index. However, these results would appear unexpected to a user. Table 4.3 shows a few examples from applying the Porter stemming algorithm. In most cases, the root word obtained is not an English word. Of particular interest, the word “skies” is transformed to the unrelated word “ski”, instead of “sky”, through the removal of the ending “es”. Nevertheless, the benefits of stemming outweigh the costs, and provided all related words and no unrelated words stem to the same string most of the time, no problems are caused. As words are combined to form longer words during browsing, it is expected that neighbouring words make up for any important contexts lost during stemming.

The stemming algorithm used by RE-PAIR contains minor changes to the original stemming algorithm by Porter. Details of the changes are provided in Appendix B. The American spelling assumed by the implementation is retained in order to accommodate the spelling used by the test data.

As with case folding, compression implies a requirement to be able to losslessly reverse any stemming performed. A stream of stemming modifiers is generated, with each word token having a corresponding modifier. Appendix B lists the steps to the algorithm and shows that a modifier of length 23 bits is sufficient to indicate how to unstem a word. Each step (and sub-step) is assigned bit positions in the modifier, independent of other steps. While it might have been possible to reduce the number of bits used by the modifier
Selecting Phrases for Browsing

by combining steps, separating the steps keeps the variant used by RE-PAIR as close to the original as possible.

Since stemming reduces the length of a word, an alternative design for PRE-PAIR would apply the 16 character limit on word tokens after stemming. However, this approach was not taken for two reasons. First, PRE-PAIR would be less efficient because a buffer of characters which varies in length would have to be maintained. Second, the one-to-one correspondence between word tokens and stemming modifiers could no longer be guaranteed. Reversing stemming not only increases the length of a stem, but changes characters within it as well. So, if a long word is split across two word tokens, it may be incorrect to assign the stemming modifier to only the second word token. The consequence of this decision is that words longer than 16 characters would not be stemmed, even if their stemmed form is less than the limit.

There are two reasons for incorporating case folding and stemming into word-based RE-PAIR. During phrase browsing, these transformations group words that are derived from the same root word to improve usability, as shown later in Chapter 7 when the phrase browser is presented. Also, both case folding and stemming help reduce the number of entries in the word lexicon. The effect these two transformations have on the number of distinct word tokens is shown in Table 4.4 for the test file WSJ20. A detailed discussion of the experiments with WSJ20, including compression times and compression ratios, is provided later, in Section 4.5. Some of those results are presented here to complete the discussion on case folding and stemming. Case folding and stemming do not have to be used together. However, because the implementation of the Porter stemming algorithm used by word-based RE-PAIR only stems lower case words; words with endings in upper case are untouched. This occurred often in WSJ20 since certain sections of every news article was entirely in upper case.

Lexicon statistics when the application of case folding and stemming is varied are shown in Table 4.4. In all cases, PRE-PAIR identified 3,356,915 word tokens. Both case folding and stemming decrease the number of unique word tokens, as shown in the first column. The average length of a word token in the message is 4.92 characters. Note that
4.3 Word-based Re-Pair

<table>
<thead>
<tr>
<th></th>
<th>Unique tokens</th>
<th>Average length in lexicon (chars)</th>
</tr>
</thead>
<tbody>
<tr>
<td>word lexicon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>no transformations</td>
<td>74,686</td>
<td>7.31</td>
</tr>
<tr>
<td>case folding only</td>
<td>60,748</td>
<td>7.38</td>
</tr>
<tr>
<td>stemming only</td>
<td>50,412</td>
<td>6.26</td>
</tr>
<tr>
<td>case folding and stemming</td>
<td>39,638</td>
<td>6.27</td>
</tr>
<tr>
<td>non-word lexicon</td>
<td>1,399</td>
<td>7.68</td>
</tr>
</tbody>
</table>

Table 4.4: Statistics from Pre-PAIR which indicate the effects of case folding and stemming on the word lexicon. Transformations to word tokens do not affect the non-word lexicon, which is also shown. The test data used was WSJ20.

This value is smaller than the average token lengths in the lexicons, as shown in the second column of Table 4.4. As one would expect, in an English document, short words occur more frequently than long ones. The small averages imply that extending the limit of 16 characters per token would achieve little benefit for document collections such as WSJ20.

Case folding and stemming transformations have no effect on the non-word lexicon, whose size is shown in the last row of Table 4.4. The number of entries in the non-word lexicon is much less than the word lexicon. The average length of a non-word token in the message is 1.34 characters, much less than the 7.68 characters in the non-word lexicon.

While modifiers for case folding and stemming can be stored together, they have been separated and placed into their own individual streams. This separation ensures a one-to-one correspondence between a word token and its associated modifiers. This approach highlights an important point regarding how modifiers are used. While users usually search for words, it would be difficult to envision a user searching for a particular case folding modifier. Instead, when a user has found a word token of interest, the position $k$ of the word token in the message needs to be determined. Then, to reverse case folding and stemming, the system must retrieve the modifiers at position $k$ in the respective streams. Experiments with several compression systems were applied later in this chapter in order to process the two streams in the context of this requirement. But first, the next section provides more details about the two lexicons.
Selecting Phrases for Browsing

Encoding the Lexicons

After a word token is case folded and stemmed, it is located in the word lexicon and replaced with an ordinal number indicating its position in the lexicon. If the word token is novel, then it is replaced with the number representing the next available position, and the lexicon is updated. The pointer back to the lexicon is appended to the output word sequence. After PRE-PAIR completes, its lexicon is written as a separate file while the stream of word sequence numbers is then passed to RE-PAIR, to produce the reduced word sequence. Finally, the reduced word sequence is compressed, to yield the compressed word sequence.

In contrast to the word sequence, the stream of non-word tokens are processed without any stemming, case folding, or application of RE-PAIR. Also, the non-word sequence is called the *non-word modifier* stream. That is, the symbols in this stream are considered modifiers which transform word tokens by appending non-word symbols to them.

The PRE-PAIR system keeps each lexicon in memory with a splay tree [Sleator and Tarjan, 1985]. A splay tree is a binary search tree that is rotated after every node access or modification to ensure that the most recently used node is at the root of the tree. The number of entries in the lexicon is equal to the number of unique case folded, stemmed words in the document. Heinz et al. [2002] describes an alternative data structure for maintaining lexicons called a *burst trie*.

Three methods of compressing the lexicons exist. The first method involves the application of a transformation called *front coding* [Gottlieb et al., 1975]. Front coding is a technique often used to encode a sorted list of strings. Many adjacent strings in a sorted list have similar prefixes. Front coding exploits this fact by encoding every word in the lexicon as a triple made up of two numbers and a substring. The first number indicates the number of characters the prefix of the current word matches to the prefix of the previous word (0 if nothing matches). The second number is the remaining length of the current word, without the length of the matching prefix. The substring of the remainder of the word is the last part of the triple. Recall that in word-based RE-PAIR, word and non-word tokens are limited to 16 characters in length. Therefore, 8 bits are sufficient for indicating
4.3 Word-based Re-Pair

the two numeric triples of each word.

The second method of lexicon encoding uses one of the compression systems described in Chapter 2. The lexicon is written in sorted order, with each token preceded by one byte, indicating its length. In the experiments described shortly, this is called literal encoding. Then one of the following three compression systems are applied: GZIP, BZIP2, and PPMD. Similar to the experiments in the previous chapter, GZIP and BZIP2 were executed using the -9 option in order to favour compression effectiveness. Likewise, PPMD used 255 MiB of memory and a seventh order context. The third scheme combines the first two by applying one of these compression systems to the output of the front coder. Nevill-Manning and Witten [1997] also used PPM after front coding for compressing SEQUITUR's word lexicon.

Table 4.5 and Table 4.6 show the compression effectiveness of the word lexicons with respect to itself and with respect to the size of WSJ20, respectively. Case folding and stemming have been varied along with the three compression strategies. The two views of the results are necessary because these transformations affect the size of the lexicons. As a benchmark, the tables also provide compression ratios when literal coding is used without any additional compression system. Compression ratios are reported in bits per character (bpc) with the best result in each table highlighted.

When literal coding is not coupled with any compression mechanism, the size of the word lexicons are shown in the tables in the rows marked "none", under the first columns. Case folding and stemming both decrease the amount of space occupied. In all four combinations, GZIP achieves the best compression when no front coding is applied. Of course, a smaller word lexicon means that some compression effectiveness must be paid for by additional modifier streams, as described later in this chapter. Once front coding is used, compression with any of the three systems surveyed is better than applying the same program by itself. In particular, in all cases, front coding with BZIP2 offers the best compression effectiveness.

Table 4.7 shows the effect of applying these compression mechanisms on the non-word lexicon. On the left, the compression results are reported with respect to the size of the lexicon; on the right, they are presented with respect to the size of WSJ20. According to
Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>8.000</td>
<td>3.318</td>
</tr>
<tr>
<td>GZIP</td>
<td>3.411</td>
<td>2.054</td>
</tr>
<tr>
<td>BZIP2</td>
<td>3.770</td>
<td>1.771</td>
</tr>
<tr>
<td>PPMD</td>
<td>3.442</td>
<td>1.864</td>
</tr>
</tbody>
</table>

(a) No case folding or stemming

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>8.000</td>
<td>3.958</td>
</tr>
<tr>
<td>GZIP</td>
<td>3.576</td>
<td>2.512</td>
</tr>
<tr>
<td>BZIP2</td>
<td>4.233</td>
<td>2.202</td>
</tr>
<tr>
<td>PPMD</td>
<td>3.926</td>
<td>2.363</td>
</tr>
</tbody>
</table>

(c) Stemming only

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>8.000</td>
<td>3.801</td>
</tr>
<tr>
<td>GZIP</td>
<td>3.701</td>
<td>2.341</td>
</tr>
<tr>
<td>BZIP2</td>
<td>4.130</td>
<td>2.085</td>
</tr>
<tr>
<td>PPMD</td>
<td>3.905</td>
<td>2.226</td>
</tr>
</tbody>
</table>

(d) Case folding and stemming

Table 4.5: Comparing the use of front coding with several compression systems on the word lexicon of WSJ20 while varying the use of case folding and stemming. Results are in bits per character with respect to the size of the lexicon.

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0.237</td>
<td>0.104</td>
</tr>
<tr>
<td>GZIP</td>
<td>0.101</td>
<td>0.061</td>
</tr>
<tr>
<td>BZIP2</td>
<td>0.112</td>
<td>0.052</td>
</tr>
<tr>
<td>PPMD</td>
<td>0.102</td>
<td>0.055</td>
</tr>
</tbody>
</table>

(a) No case folding or stemming

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0.194</td>
<td>0.082</td>
</tr>
<tr>
<td>GZIP</td>
<td>0.080</td>
<td>0.047</td>
</tr>
<tr>
<td>BZIP2</td>
<td>0.090</td>
<td>0.041</td>
</tr>
<tr>
<td>PPMD</td>
<td>0.083</td>
<td>0.042</td>
</tr>
</tbody>
</table>

(b) Case folding only

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0.140</td>
<td>0.069</td>
</tr>
<tr>
<td>GZIP</td>
<td>0.067</td>
<td>0.044</td>
</tr>
<tr>
<td>BZIP2</td>
<td>0.074</td>
<td>0.038</td>
</tr>
<tr>
<td>PPMD</td>
<td>0.068</td>
<td>0.041</td>
</tr>
</tbody>
</table>

(c) Stemming only

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0.110</td>
<td>0.062</td>
</tr>
<tr>
<td>GZIP</td>
<td>0.061</td>
<td>0.032</td>
</tr>
<tr>
<td>BZIP2</td>
<td>0.067</td>
<td>0.029</td>
</tr>
<tr>
<td>PPMD</td>
<td>0.054</td>
<td>0.031</td>
</tr>
</tbody>
</table>

(d) Case folding and stemming

Table 4.6: Comparing the use of front coding with several compression systems on the word lexicon of WSJ20 while varying the use of case folding and stemming. Results are in bits per character with respect to the size of WSJ20.

the table on the right, the non-word lexicon is only 0.005 bpc when output as literal text, while the word lexicon is at least 0.110 bpc when case folding and stemming are both applied. Again, front coding with BZIP2 provided good compression effectiveness at 0.001
4.4 Punctuation-aligned Re-Pair

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>8.000</td>
<td>2.925</td>
</tr>
<tr>
<td>GZIP</td>
<td>2.632</td>
<td>1.464</td>
</tr>
<tr>
<td>BZIP'2</td>
<td>2.588</td>
<td>1.446</td>
</tr>
<tr>
<td>PPMD</td>
<td>2.385</td>
<td>1.373</td>
</tr>
</tbody>
</table>

(a) Relative to the lexicon

<table>
<thead>
<tr>
<th></th>
<th>Literal coding</th>
<th>Front coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0.005</td>
<td>0.002</td>
</tr>
<tr>
<td>GZIP</td>
<td>0.002</td>
<td>0.001</td>
</tr>
<tr>
<td>BZIP'2</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>PPMD</td>
<td>0.001</td>
<td>0.001</td>
</tr>
</tbody>
</table>

(b) Relative to WSJ20

Table 4.7: Comparing the difference between the use of front coding with several compression systems on the non-word lexicon of WSJ20. Results are in bits per character.

Bye. Other regimes are also able to attain similar compression ratios for the non-word lexicon. These tables do not mention the time required for compressing the lexicons. Later in this chapter, detailed experiments with WSJ20 show that both lexicons are small in size compared to the other streams. Processing the lexicons contributes little to the overall compression and decompression times.

4.4 Punctuation-aligned Re-Pair

The pre-processing step of PRE-PAIR solves many of the problems that existed with character-based RE-PAIR and word-aligned RE-PAIR. A word in the message is no longer broken into more than one phrase. Moreover, all words, even infrequent ones, are in the word lexicon and are therefore primitives in the phrase hierarchy.

One minor problem remains, which is related to the treatment of punctuation marks. In English, punctuation marks such as commas and full stops separate phrases or sentences from each other. Word-based RE-PAIR ignores these punctuation marks and allows phrases to span over them. A minor change to word-based RE-PAIR, called punctuation-aligned RE-PAIR, encourages phrases to end with punctuation marks.

Punctuation-aligned RE-PAIR identifies a set of punctuation marks as special processing symbols. The set of six punctuation marks are: the exclamation mark (!), the full stop (.), the question mark (?), the comma (,), the colon (:), and the semicolon (;). The first three punctuation marks set off a sentence from the surrounding text, while the remaining three separate an independent clause, a phrase, or a subordinate clause from the rest of the
sentence. During parsing with PRE-PAIR, these punctuation marks are identified without understanding the context in which they occur. Because of this, PRE-PAIR treats a full stop in an abbreviation as if a sentence had just ended. All of these punctuation marks are categorised as *separating punctuation marks*, in contrast to *enclosing punctuation marks* [Greenbaum, 1996]. Enclosing punctuation marks, such as parentheses, highlight a section of text usually through a pair of delimiters.

While other punctuation marks could have been identified, adding too many would have degraded the performance of PRE-PAIR. Moreover, Greenbaum [1996] presented results from a study of American texts which consisted of writing totalling 72,000 words. The documents were all samples of “journalism, learned writing, and fiction”. The results showed that the six punctuation marks used by punctuation-aligned RE-PAIR accounted for 95.9% of all punctuation marks. Full stops and commas alone contributed to 91.8% of the punctuation marks found. In comparison, these six punctuation marks make up 98.0% and 61.9% of the punctuation marks in bible.txt (Large Canterbury Corpus, described in Chapter 2) and WSJ20. The percentage of punctuation marks for WSJ20 is lower, even though this value has already excluded SGML tags. A few lines from WSJ20 were provided in Figure 2.7(a) on page 40. As the figure shows, other punctuation marks such as hyphens and slashes appear, which do not segment sequences of words into phrases.

In the same way that word-aligned RE-PAIR modifies character-based RE-PAIR, punctuation-aligned RE-PAIR augments a set of rules to word-based RE-PAIR which dictate what active pairs can be chosen. These rules are shown in Table 4.8. Every symbol in the word sequence is assigned a category. Either it is a word that is not followed by any of the 6 punctuation marks (W), or it is (P). The table gives the four ways these two symbol types, shown under the first two columns of the table, can combine. If the combination is permitted, and subsequently replaced, the new symbol obtains a category as shown in the third column. The fourth column of the table gives some examples in which the symbol combinations may occur using variations on “Melbourne University”, an unofficial name for The University of Melbourne.

Whether or not a word is followed by a punctuation mark, it can become an active
4.4 Punctuation-aligned Re-Pair

<table>
<thead>
<tr>
<th>Left</th>
<th>Right</th>
<th>Result</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>W</td>
<td>W</td>
<td>Melbourne University</td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>P</td>
<td>MJU:</td>
</tr>
<tr>
<td>W</td>
<td>P</td>
<td>P</td>
<td>Melbourne University.</td>
</tr>
<tr>
<td>P</td>
<td>W</td>
<td>Not allowed</td>
<td>In Melbourne, universities</td>
</tr>
</tbody>
</table>

Table 4.8: The rules used by punctuation-aligned Re-Pair for deciding when a pair of symbols can become active pairs. An active pair is allowed if any of the first three rules apply; an active pair is disallowed if the last rule applies.

pair with symbols of the same category. If it is replaced, the new symbol inherits the same category as its components. The rule in the second row permits abbreviations and words in a series, separated by commas, to form. However, symbols of two different types can only combine if the right component is a symbol followed by a punctuation mark. The reversed combination (fourth row in the table) is disallowed. Note that these rules do not pair symbols based on the categories of the surrounding symbols. While word-aligned Re-Pair forced complete words and non-words to be formed by examining the contexts in which symbols occur, punctuation-aligned Re-Pair is less restrictive. For example, in the second rule of Table 4.8, two symbols with P categories can combine regardless of the classification of the symbol to the left of the left component. Punctuation-aligned Re-Pair merely encourages word tokens to combine to the left if they are followed by punctuation marks. This design choice recognises the fact that phrases can form without punctuation marks, so confining phrases to exist only between punctuation marks would be inappropriate.

There are three other differences between word-aligned and punctuation-aligned Re-Pair. First, a mixture (M) category is unnecessary since anything that combines with a word token terminated by a punctuation mark is a (P) token. Second, the symbol categories are stored with the sequence and not with the phrase hierarchy. Unlike word-aligned Re-Pair, the category of a word token depends entirely on the presence of a punctuation mark after it. Therefore, maintaining the categories in the phrase hierarchy is not possible. As a result, for a message of n symbols, n extra words of memory are required to implement punctuation-aligned Re-Pair. Since only one bit is ever used, this
approach is wasteful, and a more careful implementation is possible which operates on the bit level. Then, only $n$ extra bits of memory would be needed. Finally, the information about punctuation marks has been separated from the word tokens by PRE-PAIR. In order to implement punctuation-aligned RE-PAIR, punctuation flags (shown as dashed lines in Figure 4.3 on page 82) need to be passed to RE-PAIR along with the word sequence. For each word token, the presence or absence of a punctuation mark following it is indicated by a flag. These flags are only used by RE-PAIR, and the output structure of punctuation-aligned RE-PAIR remains identical to that of word-based RE-PAIR.

Figure 4.4 shows the difference between the four versions of RE-PAIR with respect to the type of phrases created. The document used is bible.txt, chosen because the SGML markup in the NEWS file, and the files that are derived from it, would clutter the example. After being compressed, the first 20 symbols have been expanded into primitives for each version of RE-PAIR. Since character-based and word-aligned RE-PAIR do not remove non-word characters, the space and linefeed characters are represented as "·" and "#", respectively, in the figure. Case folding and stemming were used for word-based and punctuation-aligned RE-PAIR, with a space added between each word to facilitate display.

Since character-based RE-PAIR forms phrases using only frequency, word boundaries are not adhered to, as shown in Figure 4.4(a). For example, in the fourth line, the word "waters" spans the first and the second phrases of that line. Also, some phrases commence with non-words. Word-aligned RE-PAIR handles both of these problems by using its rules to guide the recursive pairing. While a phrase may contain part of a word, no word spans across phrases, as shown in Figure 4.4(b). Also, all phrases start with a complete word and end with a complete non-word. Alternatively, they can also contain all word characters, or all non-word characters. A phrase categorised as a mixture cannot begin with a non-word character.

The phrases from word-based RE-PAIR in Figure 4.4(c) have been case folded and stemmed. Since the smallest atomic units are word tokens, word-based RE-PAIR ensures that no word spans across one or more phrases. Moreover, non-word characters do not appear in the expanded phrases anymore. While case folding and stemming have
4.4 Punctuation-aligned Re-Pair

In the beginning, God created the heaven and the earth.
And the earth was without form, and void; and darkness was upon the face of the deep.
And the Spirit of God moved upon the face of the water. And God said, Let there be light;
and there was light.

(a) Character-based Re-Pair

In the beginning, God created the heaven and the earth.
And the earth was without form, and void; and darkness was upon the face of the deep.
And the Spirit of God moved upon the face of the water. And God said, Let there be light;
and there was light.

(b) Word-aligned Re-Pair

in the begin god creat the heaven and the earth
and the earth was without form and void and dark was
upon the face of the deep and the spirit of god move
upon the face of the water and god said let there be light
and there was light and god saw the light that it was

(c) Word-based Re-Pair with case folding and stemming

in the begin god creat the heaven and the earth
and the earth was without form and void and dark was
upon the face of the deep and the spirit of god move
upon the face of the water and god said let there be light
and there was light and god saw the light

(d) Punctuation-aligned Re-Pair with case folding and stemming

Figure 4.4: Expanding the first 20 symbols of bible.txt into primitives, after being compressed with character-based Re-Pair, word-aligned Re-Pair, word-based Re-Pair, and punctuation-aligned Re-Pair. For the purposes of display, • and \_ are used to indicate linefeeds and spaces, respectively, in (a) and (b). Non-words are not shown in (c) and (d).

transformed some of the words, they still resemble the original words, when compared with Figure 4.4(a). Word-based Re-Pair ignores the location of punctuation marks. Punctuation-aligned Re-Pair rectifies this problem by encouraging word tokens followed by a punctuation mark to only pair with a symbol to its left. For example, comparing word-based Re-Pair with punctuation-aligned Re-Pair, in the second line of Figure 4.4(d), “without form, and void” is no longer accepted as a phrase. Also, while word-based Re-Pair created the phrases “and god said let there be” and “light” in the fourth line, punctuation-aligned Re-Pair separated these seven words according to the locations of
### Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th></th>
<th>Symbol range</th>
<th>Distinct values</th>
<th>Number of symbols</th>
<th>Self-information</th>
</tr>
</thead>
<tbody>
<tr>
<td>case folding modifier</td>
<td>0 to 65,536</td>
<td>72</td>
<td>3,356,915</td>
<td>1.227</td>
</tr>
<tr>
<td>stemming modifier</td>
<td>0 to 7,356,416</td>
<td>346</td>
<td>3,356,915</td>
<td>2.292</td>
</tr>
<tr>
<td>non-word modifier</td>
<td>0 to 1,398</td>
<td>1,399</td>
<td>3,356,915</td>
<td>1.750</td>
</tr>
<tr>
<td>word sequence</td>
<td>0 to 39,637</td>
<td>39,638</td>
<td>3,356,915</td>
<td>10.272</td>
</tr>
<tr>
<td>reduced word sequence</td>
<td>0 to 235,929</td>
<td>218,822</td>
<td>1,582,065</td>
<td>15.829</td>
</tr>
</tbody>
</table>

Table 4.9: Statistics about some of the streams from Pre-Pair, and the reduced word sequence from Re-Pair on the WSJ20 file. The column headed self-information is the zero-order self-information for the frequency distribution of the symbols in that file, measured for each file in bits per symbol relative to the size of that file.

The comma and the colon.

### 4.5 Experiments

The compression efficiency and effectiveness of the three methods of word-aware document parsing were compared with each other and with character-based Re-Pair through experiments. Partial results from these experiments have already been shown in Section 4.3. In all of these experiments, WSJ20 was used as test data and the test machine was a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache.

Before the four versions of Re-Pair are compared, a detailed look at the various streams of word-based Re-Pair are examined. Section 4.3 showed compression results for the word and non-word lexicons of WSJ20. For both of these streams, front coding followed by BZIP2 offered the best compression effectiveness. Statistics about the remaining four streams produced by Pre-Pair are listed in Table 4.9.

The word sequence is compressed with Re-Pair, by treating the word sequence symbols as primitives. A hierarchy of phrases is constructed for browsing, as well as a reduced word sequence. In Chapter 3, experiments showed that the phrase hierarchy can be effectively stored by transforming it with chiastic slide, and then encoding it with interpolative coding. Statistics about the reduced word sequence of WSJ20 are also given in Table 4.9.

The three streams of modifiers are stored as 32-bit integers. The number of distinct symbols in any of these streams is less than the word sequence. Also the self-information
4.5 Experiments

<table>
<thead>
<tr>
<th></th>
<th>SHUFF</th>
<th>GZIP</th>
<th>BZIP2</th>
<th>PPMD</th>
<th>RE-PAIR with SHUFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>case folding modifiers</td>
<td>0.232</td>
<td>0.200</td>
<td>0.142</td>
<td>0.229</td>
<td>0.125</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td>0.388</td>
<td>0.583</td>
<td>0.457</td>
<td>0.520</td>
<td>0.394</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td>0.320</td>
<td>0.367</td>
<td>0.261</td>
<td>0.362</td>
<td>0.224</td>
</tr>
</tbody>
</table>

Table 4.10: Compression effectiveness of the modifiers from word-based RE-PAIR for WSJ20.

of these streams is lower than the word sequence.

The modifiers are also compressed, but since a phrase hierarchy is not required for browsing, compression systems other than RE-PAIR can also be applied. Results obtained by applying five compression systems to WSJ20 are presented in Table 4.10. The compression mechanisms employed include the zero-order Huffman coder SHUFF, GZIP, BZIP2, PPMD, and RE-PAIR. The SHUFF program processed the modifiers as a single block, while GZIP and BZIP2 were executed with the -9 option in order to favour compression effectiveness. The PPMD system operated with a seventh order model and a limit of 255 MiB of memory. The three modifiers were also processed with RE-PAIR as a single block, with SHUFF used to encode the sequence, also as a single block. While GZIP, BZIP2, and PPMD compress the modifiers as individual bytes, both RE-PAIR and SHUFF treat each 32-bit integer as a symbol.

The compression results from Table 4.10 show the compression results in bits per character, relative to the size of WSJ20. The best compression ratio for each modifier is highlighted. While the case folding and non-word modifiers are most effectively processed by RE-PAIR (with SHUFF), SHUFF alone provided the best compression ratio for the stemming modifiers. Moreover, BZIP2 also gave better compression effectiveness than SHUFF for the same modifiers as RE-PAIR. Both RE-PAIR and BZIP2 are better at capturing the repetition in the modifier streams.

The time required to process these streams with the five systems are reported in Table 4.11, with the fastest encoding and decoding time for each stream highlighted. The SHUFF system operated the fastest for both encoding and decoding, partly due to the simplistic zero-order model that it is based on. The BZIP2 program was particularly slow for encoding the non-word modifier, but otherwise yielded consistent encoding and
Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th></th>
<th>SHUFF</th>
<th>GZIP</th>
<th>BZIP2</th>
<th>PPMD</th>
<th>RE-PAIR with SHUFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encoding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>case folding modifiers</td>
<td>0.4</td>
<td>20.4</td>
<td>4.8</td>
<td>4.0</td>
<td>9.5</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td>0.4</td>
<td>51.0</td>
<td>4.6</td>
<td>5.0</td>
<td>13.3</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td>0.4</td>
<td>17.1</td>
<td>70.6</td>
<td>5.3</td>
<td>10.4</td>
</tr>
<tr>
<td>Decoding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>case folding modifiers</td>
<td>0.1</td>
<td>0.2</td>
<td>1.3</td>
<td>4.6</td>
<td>0.1</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td>0.2</td>
<td>0.3</td>
<td>1.9</td>
<td>5.5</td>
<td>0.3</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td>0.1</td>
<td>0.3</td>
<td>3.4</td>
<td>5.9</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 4.11: Compression times for the modifiers produced by word-based RE-PAIR on WSJ20, reported in CPU seconds and averaged over three runs each.

decoding times. Decoding with the dictionary-based systems (GZIP and RE-PAIR) are fast, and in one case, gave the same decoding time as SHUFF.

These results show that a combination of SHUFF alone and RE-PAIR coupled with SHUFF seem to offer the best compression effectiveness for the modifiers. In addition, both systems offer fast decoding times, an important factor to consider when designing a retrieval system. While these two systems satisfy the needs of compression, they are inappropriate for retrieval for other reasons. These reasons are related to how these streams are used in the browsing system, a problem which is covered later beginning in Chapter 6. The method used for encoding the reduced sequence and the modifiers is revisited at that time.

Table 4.12 presents some statistics when RE-PAIR is applied to either the original WSJ20 file of 20,971,520 characters (character-based and word-aligned RE-PAIR) or to the word sequence of 3,356,915 symbols (word-based and punctuation-aligned RE-PAIR) produced from PRE-PAIR. The results for character-based RE-PAIR are taken from Table 3.4 on page 63. Word-aligned RE-PAIR produces a larger phrase hierarchy than character-based RE-PAIR, but with less generations and a longer reduced sequence. Many phrases in the word-aligned phrase hierarchy include a phrase "α" in one generation, and then the phrase "α", in the next generation. This effect is also evident in the phrase hierarchy of Figure 4.2 on page 80.

As Table 4.12 shows, word-based and punctuation-aligned RE-PAIR are similar, but
4.5 Experiments

| Method              | $|\Sigma|$ | $|ho|$ | $|S|$ | Generations | Longest phrase (in primitives) |
|---------------------|-----------|--------|-------|-------------|-------------------------------|
| character-based     | 90        | 310,486| 1,904,118| 21          | 1,257                         |
| word-aligned        | 90        | 343,520| 1,912,566| 19          | 1,269                         |
| word-based          | 39,638    | 196,291| 1,582,065| 15          | 215                           |
| punctuation-aligned | 39,638    | 191,924| 1,634,652| 13          | 215                           |

Table 4.12: Statistics from experiments with the four variants of RE-PAIR with WSJ20. The first two methods are applied to WSJ20 directly, where the characters are primitives. The other two methods are used on the word sequence produced by PRE-PAIR.

both produce a shorter reduced sequence and a smaller phrase hierarchy than character-based and word-aligned RE-PAIR. Also, the maximum phrase generation between the two is 15. The identification of word tokens with PRE-PAIR allow contexts of characters to be found during pre-processing, before RE-PAIR is used.

The last column of the table indicates the longest phrase of each method, measured in primitives. The longest phrase for word-based RE-PAIR is 215 symbols, which expands to 1,325 characters in length. This is a measure of stemmed, case folded words with no non-words. Without these two transformations, the same phrase also has more primitives than the longest phrase for character-based RE-PAIR: 1,421 characters.

The compression effectiveness of the four versions of RE-PAIR is listed in Table 4.13. The test file is WSJ20 and compression ratios are given in bits per character relative to the original file. The results for character-based RE-PAIR are from Table 3.5 on page 64. Compression levels for four variants of word-based RE-PAIR are presented, which each differ by whether or not case folding and stemming are applied. Punctuation-aligned RE-PAIR imposes rules on RE-PAIR which prevents certain phrases from forming. However, the three modifier streams are unchanged. In the table, only results for punctuation-aligned RE-PAIR with both case folding and stemming are shown. All lexicons are encoded with front coding followed by BZIP2. The reduced word sequence and the three streams of modifiers are encoded using SHUFF.

The best compression achieved is by word-based RE-PAIR, when no word transformations are applied. Compression levels degrade as case folding and stemming are included. Note that the rules for stemming assume that the word tokens have been case folded.
<table>
<thead>
<tr>
<th>Method</th>
<th>Word lexicon</th>
<th>Phrase hierarchy</th>
<th>Reduced (word) sequence</th>
<th>Case folding modifiers</th>
<th>Stemming modifiers</th>
<th>Non-word lexicon</th>
<th>Non-word modifiers</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>character-based</td>
<td>1.774</td>
<td>0.223</td>
<td>1.551</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>word-aligned</td>
<td>1.734</td>
<td>0.201</td>
<td>1.533</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>word-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no transformations</td>
<td>0.052</td>
<td>0.154</td>
<td>1.320</td>
<td>-</td>
<td>-</td>
<td>0.001</td>
<td>0.320</td>
<td>1.708</td>
</tr>
<tr>
<td>case folding only</td>
<td>0.041</td>
<td>0.152</td>
<td>1.285</td>
<td>0.232</td>
<td>-</td>
<td>0.001</td>
<td>0.320</td>
<td>2.031</td>
</tr>
<tr>
<td>stemming only</td>
<td>0.038</td>
<td>0.145</td>
<td>1.264</td>
<td>-</td>
<td>0.380</td>
<td>0.001</td>
<td>0.320</td>
<td>2.148</td>
</tr>
<tr>
<td>both</td>
<td>0.029</td>
<td>0.143</td>
<td>1.230</td>
<td>0.232</td>
<td>0.388</td>
<td>0.001</td>
<td>0.320</td>
<td>2.343</td>
</tr>
<tr>
<td>punctuation-aligned</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>both</td>
<td>0.029</td>
<td>0.138</td>
<td>1.254</td>
<td>0.232</td>
<td>0.388</td>
<td>0.001</td>
<td>0.320</td>
<td>2.362</td>
</tr>
</tbody>
</table>

Table 4.13: Comparison of the compression ratios achieved by the four versions of Re-Pair for WSJ20, in bits per character.
Hence, the stemming modifiers occupy 0.380 bpc or 0.388 bpc, depending on whether or not the word tokens have been case folded. When both transformations are used, a compression effectiveness of 2.343 bpc is attained. Even if the best selection of compression mechanisms is chosen for the modifiers (see the highlighted numbers in Table 4.10), the compression ratios drop by 0.203 bpc to 2.140 bpc. So, despite the fact that word-based RE-PAIR and punctuation-aligned RE-PAIR produce a smaller phrase hierarchy, character-based RE-PAIR and word-aligned RE-PAIR achieve better compression effectiveness.

One reason for the poorer compression levels of word-based and punctuation-aligned RE-PAIR is that separating the message into several independent streams disturbs the contexts in the message. One possible direction for future work is to encode the modifier streams more carefully. Symbols in these streams do not appear independently of neighbouring symbols, and are even conditioned based on symbols from another stream. For example, if the first letter of a word is in upper case, the word is probably the beginning of a sentence. One would expect the few words that follow to be in lower case. Moreover, one of the non-word characters that immediately precede it is probably a sentence ending punctuation mark. And, since the stems of words correspond to their parts-of-speech (for example, "-ing" is associated with verbs), and these categories restrict the order of words in a sentence, it is expected that some correlation exists between stemming modifiers. This observation means that the encoding of the symbol in position $k$ of the case folding modifiers can be conditioned by preceding symbols as well as symbols in the other modifier streams, or the word sequence.

Table 4.13 also shows that word-aligned RE-PAIR achieves slightly better compression than character-based RE-PAIR, even though the former adds more phrases to the phrase hierarchy. This result shows that character-based RE-PAIR's use of frequency alone as a heuristic does not always choose the best phrases. A heuristic which forms complete words before combining with complete non-words to the right improves compression effectiveness over character-based RE-PAIR.

The compression effectiveness of the four versions of RE-PAIR are presented in Figure 4.5 as a graph, with the compression results for WSJ20 with GZIP, BZIP2, and PPMD
Selecting Phrases for Browsing

(see Table 2.5 on page 41) indicated as horizontal lines. As a further guide, but not shown in the figure, Isal and Moffat [2001] used a word-based block sorting mechanism and a Huffman coder on WSJ20 and reported compression effectiveness of 1.69 bits per character. Word-based and punctuation-aligned RE-PAIR both performed case folding and stemming. All four versions of RE-PAIR have been broken down by stream. Word-based and punctuation-aligned RE-PAIR both achieve slightly worse compression than BZIP2, but better than GZIP. However, both of these versions of RE-PAIR require more memory than either BZIP2 or GZIP. Even when neither case folding nor stemming are used with word-based RE-PAIR, the compression effectiveness of 1.708 is still slightly worse than the 1.656 bpc offered by PPMD.

Other than compression effectiveness, the second factor to consider when evaluating compression systems is execution time. All times reported below have been averaged over 3 trials. Table 4.14 displays the encoding and decoding times of character-based RE-PAIR (from Table 3.6 on page 64) and word-aligned RE-PAIR. The rules embedded within word-aligned RE-PAIR reduces the number of active phrases and active pairs that character-based RE-PAIR keeps. As a result, word-aligned RE-PAIR creates a longer reduced sequence, but encodes the message in less time. However, no difference exists for the other program execution times shown in Table 4.14.

The time required to compress WSJ20 by word-based RE-PAIR, with case folding and stemming, is shown in Table 4.15. If both case folding and stemming are not used, the parsing performed by PRE-PAIR reduces to 7.8 seconds. Since the only difference between word-based RE-PAIR and punctuation-aligned RE-PAIR is the way phrases are selected, the only times affected are those related to RE-PAIR. Punctuation-aligned RE-PAIR takes 16.3 and 1.1 seconds to encode with RE-PAIR and SHUFF and 0.9 and 0.3 seconds to decode with DES-PAIR and SHUFF. Similar to word-aligned RE-PAIR, the augmented rules to punctuation-aligned reduces the amount of time to encode the word sequence.

Finally, Figure 4.6 shows the difference in phrases between word-aligned RE-PAIR and word-based RE-PAIR by expanding the first 20 symbols of WSJ20 into primitives. As a reference point, the same section of WSJ20 was presented in Figure 3.5 on page 63.
Punctuation-aligned Re-Pair is not shown because, for this section of text, only a small difference with word-based Re-Pair exists. An extensive comparison of phrases between the four versions of Re-Pair was discussed earlier with bible.txt (Figure 4.4). Figure 4.6 presents the phrases for WSJ20 for completeness, since most of this thesis uses WSJ20, or files similar to WSJ20, as test data.

The same comparisons that were made for Figure 4.4 also apply to Figure 4.6. However, a couple of points of interest exists in Figure 4.6. Note that SGML tags are treated
### Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th>Character-based</th>
<th>Word-aligned</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Encoding time</strong></td>
<td></td>
</tr>
<tr>
<td>Re-Pair</td>
<td>Re-Pair</td>
</tr>
<tr>
<td>Re-Pair</td>
<td>88.5</td>
</tr>
<tr>
<td>Shuff</td>
<td>1.6</td>
</tr>
<tr>
<td>Total</td>
<td>90.1</td>
</tr>
<tr>
<td><strong>Decoding time</strong></td>
<td></td>
</tr>
<tr>
<td>Shuff</td>
<td>2.5</td>
</tr>
<tr>
<td>Des-Pair</td>
<td>0.4</td>
</tr>
<tr>
<td>Total</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Table 4.14: Compression times for character-based Re-Pair and word-aligned Re-Pair for WSJ20. Times have been averaged over three trials and are given in CPU seconds.

<table>
<thead>
<tr>
<th>Stream</th>
<th>Compression system</th>
<th>Encoding time</th>
<th>Decoding time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Pair</td>
<td>front coding and BZIP2</td>
<td>14.2</td>
<td>2.8</td>
</tr>
<tr>
<td>word lexicon</td>
<td>Re-Pair or Des-Pair</td>
<td>19.6</td>
<td>0.9</td>
</tr>
<tr>
<td>word sequence</td>
<td>Shuff</td>
<td>1.1</td>
<td>0.2</td>
</tr>
<tr>
<td>case folding modifiers</td>
<td>Shuff</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td>Shuff</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>non-word lexicon</td>
<td>front coding and BZIP2</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td>Shuff</td>
<td>0.3</td>
<td>0.1</td>
</tr>
<tr>
<td>Total</td>
<td>36.0</td>
<td>4.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.15: Compression and decompression times for word-based Re-Pair in CPU seconds. Case folding and stemming have been performed and front coding followed by BZIP2 is used for encoding the lexicons. Times have been averaged over three trials.

as word tokens, and all of them have been case folded. Also, the word “telecommunications” on line 5 of Figure 4.6(b) is too long and has been broken in two. However, after breaking it, no stemming rules could be applied, so this word appears unchanged. If no maximum word length was imposed, the word “telecommunications” would have stemmed to “telecommun”.

104
4.6 Phrases for Browsing

Three additional heuristics for selecting phrases have been proposed in this chapter which extend the capabilities of character-based Re-PAIR. Experiments have been performed which measured the difference in compression effectiveness achieved and compression times. However, the initial motivation for embarking on this investigation has been to create phrases that are suitable for phrase browsing.

From Figure 4.4 and Figure 4.6, the most visually appealing phrases are those of word-based Re-PAIR and punctuation-aligned Re-PAIR. Case folding and stemming reduce the size of the lexicon and allow users to browse phrases without considering the case and stems used in the document. The separation of non-words from the message permits browsing without the clutter caused by them. While not appearing in the phrase hierarchy, non-word tokens which contain punctuation marks are used by punctuation-aligned Re-PAIR.
### Selecting Phrases for Browsing

<table>
<thead>
<tr>
<th>Output from Pre-PAIR</th>
<th>word lexicon</th>
<th>word sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>phrase browsing</td>
<td>searching for symbols</td>
</tr>
<tr>
<td>Model</td>
<td>front coding and BZIP2</td>
<td>RE-PAIR</td>
</tr>
<tr>
<td>Coder</td>
<td>phrase hierarchy chiasic slide and interpolative coding</td>
<td>reduced word sequence SHUFF</td>
</tr>
</tbody>
</table>

(a) Words

<table>
<thead>
<tr>
<th>Output from Pre-PAIR</th>
<th>non-word lexicon</th>
<th>non-word modifiers</th>
<th>case folding modifiers</th>
<th>stemming modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>skipping to symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>front coding and BZIP2</td>
<td>SHUFF</td>
<td>SHUFF</td>
<td>SHUFF</td>
</tr>
<tr>
<td>Coder</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Non-words and modifiers

Table 4.16: The seven streams produced by Pre-PAIR and RE-PAIR. The purpose of each stream, and the compression methods employed in this chapter for these streams are also shown. Entries marked “SHUFF” are replaced in Chapter 6 with systems better suited to those specific tasks.

to guide the choice of phrases made.

There are six streams created by the message pre-processor, Pre-PAIR. After RE-PAIR compresses the word sequence, a phrase hierarchy and a reduced word sequence are created. In Table 4.16, these seven streams (in boldface), their purpose, and the method employed in this chapter for compressing them are shown. Further details about how these seven streams are used by a phrase browsing system are given in Chapter 6 and Chapter 7. In Chapter 6, the four instances of “SHUFF” are replaced by compression systems that are more suited to the intended purposes of the reduced word sequence and the three modifier streams. The encoding methods for the lexicons and the phrase hierarchy remains unchanged for the remainder of this thesis.

Phrase browsing can be performed with only the word lexicon and the phrase hierarchy to determine the words used in the message and the relationships between them. Together, these two represent just 0.167 bits per character (punctuation-aligned RE-PAIR, from
Table 4.13), or just over 2% of the original document size, yet provide a very good guide to the context of the document. The lexicons are relatively small, and can be compressed using methods other than front coding with BZIP2.

The word lexicon and word phrase hierarchy are used for browsing. But when the context in which the phrases occur are required, then the compressed word sequence (1.254 bpc) can be used. The 3 streams so far are only an approximation of the document; words in the phrases need to have the case folding and stemming reversed. Also, the non-words within phrases have to be re-inserted. If punctuation-aligned Re-PAIR is used for compression, then the original document can be obtained by decoding all of the streams and applying a post-processor to combine them. For retrieval, though, only a portion of the modifiers and the non-word sequence may be required. If a one-to-one correspondence between a word and its case folding, stemming, and non-word modifiers is ensured, then a phrase can be reproduced during retrieval by skipping to precise locations in each of these streams. In this chapter, the Huffman coding used to encode the modifiers prevent this operation from occurring without decoding the entire document. However, this issue and the problem of efficiently searching for phrase contexts in the compressed word sequence are considered later, in Chapter 6. Also, more details about how a user browses phrases and searches for the contexts in which the phrases occur is examined in Chapter 7.

But first, in the next chapter, a different problem is considered. So far, Re-PAIR has been used to compress a 20MiB document. Ideally, larger documents should be compressed and browsed and some ways of achieving this is discussed next.
Chapter 5

Block Merging for Re-Pair

Phrase browsing techniques become more important as document size increases. When character-based Re-PAIR was introduced in Chapter 3, the largest document processed by Re-PAIR was a 20 MiB document, WSJ20. Limitations on computer memory prevent significantly larger documents being handled by Re-PAIR. The space analysis for Re-PAIR (Section 3.2, from page 52) showed that its memory usage is linear to the length of the message. However, the analysis also showed that the constant factor is high. For example, a 100 MiB message would require more than 1 GiB of memory for the sequence alone.

While the goal of Pre-PAIR was to align words in a document for the purpose of browsing, both word-based and punctuation-aligned Re-PAIR also reduce the memory requirements of Re-PAIR. Pre-PAIR separates a message into multiple streams, including a word sequence stream. In the previous chapter, it was shown that the parsing of WSJ20 by Pre-PAIR yielded roughly one word token for every 5 characters in the message for Re-PAIR. The WSJ20 document of 20 million symbols (characters) is transformed into a word sequence of just over 3 million symbols (references to the word lexicon). Moreover, the shorter input into Re-PAIR has a beneficial effect on the overall compression time, even through the set of primitives increases in size. When character-based Re-PAIR is applied to WSJ20, 88.5 seconds is required (Table 3.6, page 64). However, word-based Re-PAIR requires only 19.6 seconds (Table 4.15, page 104). (These times exclude the costs in parsing and entropy coding of the sequences and modifiers, which together add 1.6 and

109
Block Merging for Re-Pair

16.4 seconds back on to the execution costs of character-based and word-based Re-Pair, respectively.)

In this chapter, the problem of using Re-Pair on longer messages is considered. The solution proposed involves an initial partitioning of the message into blocks, followed by the processing of each block by Re-Pair. The division of a message into b blocks presents problems for browsing and compression. The challenge is to provide post-processing steps for Re-Pair that merge multi-block outputs, thereby solving these problems.

The remainder of this chapter is structured as follows. The creation of blocks using Re-Pair is discussed in Section 5.1. The subsequent merging of blocks occurs in three phases. The first phase, described in Section 5.2, merges the b phrase hierarchies into a single one. Section 5.3 describes phase 2, which employs the newly formed phrase hierarchy to identify pairs of symbols for further replacement. Then, Section 5.4 outlines the third phase, which re-examines the sequence to identify pairs of symbols that have not yet been represented by any symbol in the phrase hierarchy. In order to simplify the discussion in this chapter, character-based Re-Pair is used for demonstrating the three phases. Section 5.5 explores the changes required to each of these three phases in order to accommodate punctuation-aligned Re-Pair. Section 5.6 provides experiments that demonstrate the efficiency and effectiveness of block merging. Section 5.7 describes how text can be appended to a compressed document by extending the block merging technique. Section 5.8 concludes the chapter by discussing the trade-offs that exist with block merging.

5.1 Creating Blocks

To sidestep memory limitations, many compression systems partition large messages into blocks, and then process each block independent of neighbouring blocks. The gzip compression system implements an input buffer whose size is fixed for all documents, limiting the maximum block size. On the other hand, PPMD and BZIP2 create blocks according to options supplied by the user. The PPMD implementation used as a reference point in this thesis ends the current block when the maximum amount of memory specified by the user
5.1 Creating Blocks

has been exhausted, and compression is simply restarted from a “know nothing” state. The BZIP2 system allows the user to specify one of a set of block sizes.

The BZIP2 approach is also adopted by RE-PAIR for large messages. An explicit input block size is supplied by the user, which specifies that the message be divided into $b$ blocks, with all blocks equal in size, except for the last one. Each block is then compressed independently using RE-PAIR, to produce $b$ phrase hierarchies and $b$ reduced sequences.

The forceful separation of a message into blocks for compression presents three problems for browsing and compression. First, browsing multiple phrase hierarchies simultaneously is difficult, and as cumbersome as physically browsing multiple books at the same time. Second, since RE-PAIR selects phrases from each block without any information gained through the compression of previous blocks, it is likely that partitioning a uniform file produces shorter phrases. The maximum generation of symbols in the phrase hierarchies would be less than if the file is compressed as one block. Lastly, not only does the quality of phrases worsen, but compression degrades. As the block size decreases, more blocks are produced. Across any two blocks, information is duplicated in the phrase hierarchies and in the preludes in the final entropy coded sequence.

The effect blocks have on phrase quality and compression is verified by experiments using the WSJ508 file described in Chapter 2. Recall that the file is 508 MiB in size and that the WSJ20 from earlier experiments was derived from the first 20 MiB. In the experiments, WSJ508 was compressed using character-based RE-PAIR and block sizes ranging from 1 MiB to 32 MiB. Coding of the sequence was handled by the Huffman coder, SHUFF, with each block entropy coded independently of others. The results from these experiments are shown in Table 5.1 and Figure 5.1.

Table 5.1 shows that as the block size increases, the overall quality of the phrase hierarchy improves since RE-PAIR is able to find longer phrases. In the fourth column, the total number of symbols in all of the phrase hierarchies decreases as the block size increases. Finally, in the last column, the time for RE-PAIR to process the same file (excluding entropy coding), averaged over three trials, increases with the block size. Since SHUFF performs roughly the same amount of work irrespective of block size, the time taken
Table 5.1: Statistics from applying character-based Re-Pair on WSJ508 by partitioning with block sizes ranging from 1 MiB to 32 MiB. The second and third columns list the length of the longest symbol (in number of primitives) and the maximum generation of all of the blocks, respectively. The fourth column tabulates the total number of primitives and phrases across all blocks. The last column shows the execution times of Re-Pair, excluding the entropy coding of the sequence.

for coding remains the same.

Figure 5.1 depicts compression effectiveness as a function of block size. The division of the output bits between the phrase hierarchy, the compressed sequence prelude, and the compressed sequence codes are also shown. Three horizontal lines indicate the compression effectiveness of gzip, bzip2, and ppm on WSJ508, taken from Table 2.5 on page 41. Note that a comparison between these three systems and Re-Pair is unfair given that Re-Pair is using more memory. Of the four systems, gzip performs the worst, even with the -9 option, since it creates a context window of only 32 KiB. With the -9 option, bzip2 creates blocks of 900 KiB, which is comparable to the smallest 1 MiB block size of Re-Pair shown in the graph. The ppm system with 255 MiB of memory and a seventh order model achieves the best compression overall. Clearly, compression effectiveness suffers when smaller block sizes are used by Re-Pair, because symbols in the phrase hierarchy and codewords in the compressed sequence prelude are duplicated.

In order to minimise the problems with phrase browsing and compression effectiveness when Re-Pair processes a message in blocks, a block merging scheme called Re-Merge has been designed as a post-processing step. Re-Merge operates in phases, as shown in Figure 5.2. These phases can be arranged into different configurations, yielding five methods labelled method A to method E, as shown. The only required phase is phase 1, which is applied immediately after Re-Pair. Then either of two alternative second
phases is used, phase 2a or phase 2b. If phase 3 is required, it is performed last. The five combinations of these phases represent the trade-offs that exist between execution time of RE-MERGE, compression effectiveness, and phrase quality, as detailed in experiments later.

Before discussing each phase, the notation introduced in Chapter 3 needs to be extended. As before, the number of primitives and phrases in a phrase hierarchy are $|\Sigma|$ and $|\rho|$, respectively. The phrase hierarchy is denoted as $\mathcal{P}$, and the sequence as $\mathcal{S}$. The total number of primitives and phrases in the phrase hierarchy can be represented as $|\Sigma| + |\rho|$ or $|\mathcal{P}|$. The number of symbols in the sequence is $|\mathcal{S}|$. Let $\mathcal{P}_1$ and $\mathcal{P}_2$ represent initial phrase hierarchies before any phase, and $\mathcal{P}'$ represent the final phrase hierarchy after the same phase. Their corresponding sequences are $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}'$. When a symbol $\alpha$ is expanded, it becomes a string of primitives with a length equal to $|\alpha|$. In a string, a vertical bar (|)}
is used to indicate the division between the expanded left component and the expanded right component (for example, “wood|chuck”).

The next three sections describe the three phases of Re-Merge in detail.

5.2 Merging Phrase Hierarchies

The first phase of Re-Merge carries out a pair-wise merging of blocks via a series of passes. Each pass halves the number of blocks, meaning that for an initial message of \( b \) blocks, \( \lceil \log_2 b \rceil \) passes are required. During each pass, the union of the pair of phrase hierarchies, \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \), is formed to produce \( \mathcal{P}' \). After the two phrase hierarchies are merged, the sequences \( S_1 \) and \( S_2 \) are modified to ensure that each symbol refers to \( \mathcal{P}' \) and not their respective phrase hierarchies. Afterwards, \( S_1 \) and \( S_2 \) are concatenated together.
5.2 Merging Phrase Hierarchies

Initially, phase 1 decodes both phrase hierarchies. They are then merged one generation at a time, with duplicate phrases that have equal left and right components removed. Then, each symbol is expanded into a string of primitives. The phrase hierarchy is sorted by these strings, producing a temporary phrase hierarchy, $P^*$. With the strings sorted, duplicate symbols that exist across and within generations, but with different components, can also be identified and removed. Recall from page 58 that when a message (or in this case, a block) is processed by RE-PAIR, duplicates may occur within the phrase hierarchy which have different components. For example, one symbol may expand to “a|aa” while another symbol may expand to “aa|a”. The merging of two phrase hierarchies occurs in two steps in order to reduce the size of $P^*$.

The RE-PAIR decompressor, DES-PAIR, also expands phrases into primitives. However, since symbols are expanded into primitives only as needed, DES-PAIR uses a buffer of recently decoded symbols as a trade-off between execution time and memory space (Table 3.2, page 49). The needs of phase 1 of RE-MERGE differs from that of DES-PAIR. In order to sort the strings for $P^*$, ideally, all strings have to be available for sorting at once, and therefore all need to be expanded beforehand. But simply decoding every symbol could potentially occupy memory proportional to the length of the original message. Phase 1 of RE-MERGE (and subsequent phases which require the phrase hierarchy to be expanded) take an alternative approach which expands symbols starting from the last generation of symbols. Only symbols which are not used as a component of any other symbol are expanded. As a symbol is expanded, all symbols which it is composed of, either directly or indirectly, refer to the substrings of the expanded string instead. This approach ensures that a minimum amount of memory is taken up by the strings, while all of them are available for sorting and other operations in later RE-MERGE phases.

When two or more symbols in $P^*$ have been found which expand to identical strings, the one with the lower generation is retained. When a symbol has one of its components removed, it must reference the retained duplicate as a component instead. The symbol's generation number also decreases accordingly. As a result, a cascading effect occurs, and symbols that refer to a removed symbol, either directly or indirectly, shift to lower
Block Merging for Re-Pair

generations. The overall result is the reduction of gaps in the phrase hierarchy. After the removal of duplicates, the symbols in the phrase hierarchy are sorted in generation and chiastic slide value order to form \( P' \), for the interpolative coding stage.

Figure 5.3 shows an example of Re-Merge phase 1. The following message is compressed with Re-Pair using blocks of 17 characters: \underline{wood}woodywoodywoodywoodywood. Two blocks are created by Re-Pair, as shown with the underline and overline. Their respective lengths are 17 characters and 9 characters. When Re-Pair is applied to these two blocks independently, two sets of phrase hierarchies and sequences are produced, as shown in Figure 5.3(a) and Figure 5.3(b). In the example, the symbols in each of these phrase hierarchies are expanded immediately only for illustrative purposes. To the right of each expanded symbol is its generation. The two phrase hierarchies are merged so that duplicates with the same components are removed. At this stage, the algorithm expands each symbol into primitives and sorts each string, producing \( P^* \). As Figure 5.3(c) shows, the duplicate "wood" exists and the higher generation of the two is removed. This affects the fourth generation phrase "woody" by shifting it into the third generation. Figure 5.3(d) shows the final merged phrase hierarchy, \( P' \), sorted in generation and chiastic value order. Below the phrase hierarchy is \( S' \), created from the concatenation of the two initial sequences and renumbered so that it refers to the final phrase hierarchy.

In order for \( P^* \) to be formed, symbols that are not referred to by a higher generation symbol are expanded. This is depicted by Figure 5.4 for the example of Figure 5.3. In Figure 5.4, the symbols have been rearranged into a directed acyclic graph, with a level in the graph corresponding to a particular generation. The symbols with the highest generation are at the top of the figure, while primitives are lined up along the bottom. Directed edges are added from each phrase to its two components, except for the primitives which have no edges emanating from them.

The symbol expansion by Re-Merge phase 1 starts from the highest generation symbol with the greatest chiastic slide value in that generation. The equivalent in the hierarchical graph representation is to commence from the top of the graph, and then proceed toward the primitives, a level at a time. As Figure 5.4 shows, the order within a level is
5.2 Merging Phrase Hierarchies

<table>
<thead>
<tr>
<th>$\mathcal{P}_1$</th>
<th>$\mathcal{P}_2$</th>
<th>$\mathcal{P}^*$</th>
<th>$\mathcal{P}'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \rightarrow d$</td>
<td>$0 \rightarrow d$</td>
<td>$d$</td>
<td>$0 \rightarrow d$</td>
</tr>
<tr>
<td>$1 \rightarrow o$</td>
<td>$1 \rightarrow o$</td>
<td>$o$</td>
<td>$0 \rightarrow 1 \rightarrow o$</td>
</tr>
<tr>
<td>$2 \rightarrow w$</td>
<td>$2 \rightarrow w$</td>
<td>$o</td>
<td>d$</td>
</tr>
<tr>
<td>$3 \rightarrow y$</td>
<td>$3 \rightarrow y$</td>
<td>$o</td>
<td>o$</td>
</tr>
<tr>
<td>$4 \rightarrow o</td>
<td>o$</td>
<td>$4 \rightarrow o</td>
<td>d$</td>
</tr>
<tr>
<td>$5 \rightarrow w</td>
<td>oo$</td>
<td>$5 \rightarrow w</td>
<td>o$</td>
</tr>
<tr>
<td>$6 \rightarrow wo</td>
<td>d$</td>
<td>$6 \rightarrow wo</td>
<td>od$</td>
</tr>
<tr>
<td>$7 \rightarrow wood</td>
<td>y$</td>
<td>$4$</td>
<td>$wo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$wo</td>
<td>d$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$wood</td>
<td>y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$y$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

$S_1 = [5 \ 6 \ 7 \ 7]$ \hspace{1cm} $S_2 = [6 \ 3 \ 6]$ \hspace{1cm} $S' = [7 \ 8 \ 9 \ 9 \ 8 \ 3 \ 8]$

(a) \hspace{1cm} (b) \hspace{1cm} (c) \hspace{1cm} (d)

Figure 5.3: The merging of two phrase hierarchies using Re-merge phase 1. Below the two initial phrase hierarchies and the final phrase hierarchy are their respective sequences. Vertical bars (|) indicate the division between the left and right components of an expanded symbol. Also shown is the intermediate phrase hierarchy, $\mathcal{P}^*$, and the removed duplicate, “wo|d”. Symbols from every phrase hierarchy have been expanded into primitives, with each symbol’s generation shown on the right.

unimportant, since no symbol can depend on other symbols within the same generation.

With the directed graph, expansion into primitives can be summarised as first searching level-by-level to find a symbol to expand, skipping those that have already been expanded. Once a node without a directed edge leading to it has been identified, all of its descendants are “decoded” in a depth-first fashion by making each descendent refer to the higher generation node. In the example, when the fourth generation symbol, “wood|y” is first encountered, it is expanded. Then, all symbols that refer to “wood|y”, either directly or indirectly, are “expanded” by making the lower generation symbols refer to parts of the expanded string and by recording the length of the substring. Note that in this example, in order to expand the entire phrase hierarchy, only two symbols, “wood|y” and “wo|od” have to be explicitly expanded. Finally, note that after $\mathcal{P}^*$ has been sorted and duplicates removed, some symbols may be deleted even though their expansion is needed by a lower generation symbol. Because of this, it is important that symbols in $\mathcal{P}^*$ are not actually removed during phase 1, but simply ignored when $\mathcal{P}'$ is built.
Block Merging for Re-Pair

Figure 5.4: The phrase hierarchy of Figure 5.3(c), showing the two symbols that each symbol depends on. Symbols are arranged with the primitives along the bottom, and the highest generation phrase at the top. Two phrases are not used by any other symbol, “wo|od” and “wo|dy”.

The amount of memory that this phase takes is proportional to the size of the input and output phrase hierarchies. Enough space is required to keep $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}^*$ in memory. To minimise duplication in memory, the final phrase hierarchy can be drawn from $\mathcal{P}^*$. Additional space is also required for the expansion of primitives, while the concatenation and renumbering of the sequences can be performed without keeping any symbols in memory. The space requirements of phase 1 is an improvement to Re-Pair because the memory usage is proportional to the size of the phrase hierarchy, and not the length of the initial sequence.

Excluding the costs of sorting, phase 1 operates through a series of linear passes over the phrase hierarchies. First, the phrase hierarchies are merged a generation at a time by inserting symbols from $\mathcal{P}_1$ into a hash table and then comparing each symbol in $\mathcal{P}_2$ with the hash table entries. When the two initial phrase hierarchies are merged, each symbol is
5.3 Identifying Old Symbol Pairs

expanded by performing another pass over the \(|\mathcal{P}^*|\) symbols. Once the symbols are sorted, a third linear pass over \(\mathcal{P}^*\) is required to eliminate duplicates. A final sort by generation number and chiastic slide value prepares the phrase hierarchy for interpolative coding. The initial sequences are renumbered and concatenated, with buffering required only for minimizing disk accesses. Combining two phrase hierarchies requires time proportional to \(|\mathcal{P}_1| + |\mathcal{P}_2| + 2|\mathcal{P}^*| + |\mathcal{S}'|\), not including the two sorts that are performed. This amount of time is required for each of the \([\log_2 b]\) merges of a compressed message of \(b\) blocks.

The aim of phase 1 is to merge the phrase hierarchies of every block into a single, unified phrase hierarchy, while also removing any duplicate symbols. The relative sizes of the initial, intermediate, and final phrase hierarchies can be summarised as follows: 
\[
\max(|\mathcal{P}_1|, |\mathcal{P}_2|) \leq |\mathcal{P}'| \leq |\mathcal{P}^*| \leq |\mathcal{P}_1| + |\mathcal{P}_2|.
\]
After \([\log_2 b]\) iterations of phase 1, all of the phrase hierarchies have been merged and all of the sequences have been concatenated. Effectively, phase 1 has merged all of the blocks in the reduced message and the phrase hierarchy can be browsed.

While phase 1 makes phrase browsing feasible and improves compression effectiveness, even better compression can be obtained. For example, in the final sequence in Figure 5.3(d), the pair of symbols \([8, 3]\) in \(\mathcal{S}'\) can be replaced by a single symbol, \([9]\). The next section describes phase 2 of Re-Merge, which shifts the attention from the phrase hierarchy to the sequence, to try to catch such redundancies.

5.3 Identifying Old Symbol Pairs

After the phrase hierarchies have been merged, pairs of symbols may appear in the concatenated sequence that are now candidates for replacement. For example, if \(a\beta\) exists twice in block 1, but once in block 2, it would have been recognised as a phrase in the first block, but not in the second. The concatenated sequence needs to be checked to ensure no pairs of symbols exist that are already represented by symbols in the merged phrase hierarchy.

Identifying symbol pairs in the initial sequence that already have a corresponding rule for replacement is the motivation for phase 2. Phase 2 reduces the length of the
Block Merging for Re-Pair

sequence, but makes no changes to the phrase hierarchy. As with RE-PAIR, the goal of this phase is to minimise the length of the sequence. Since phase 2 is separated from the coder, no guarantees can be made that reducing the length of the sequence also improves compression effectiveness. However, phase 2 improves the usefulness of the phrase browsing interface. Suppose a user is examining a symbol $\gamma$, which expands to the pair $\alpha \beta$. Phase 2 ensures that more occurrences of this pair of symbols can be located with the phrase browsing system. Since this phase does not alter the merged phrase hierarchy $|P'| = |P_1|$. But as the aim of phase 2 is to reduce the length of the initial sequence, $|S'| \leq |S_1|$.

Two methods of performing phase 2 are explored in this section. The first method, phase 2a, is biased towards efficiency, while the second method, phase 2b, favours compression effectiveness.

Efficient Sequence Reduction

Phase 2a of RE-MERGE determines replacements by performing a single pass through the initial sequence $S_1$. As each symbol is processed in the pass, it is considered in combination with the immediately preceding symbol, and a replacement made if the pair exists in the phrase hierarchy.

The steps for phase 2a are shown in Algorithm 5.1. First, a hash table of symbols is constructed, with all phrases in the phrase hierarchy hashed according to their two components, and collisions handled by chaining. Then, each symbol in $S_1$ is processed as follows. When a symbol $\alpha_i$ from the sequence is read, it is combined with its immediately preceding symbol $\alpha_{i-1}$. The pair $\alpha_{i-1}\alpha_i$ is searched in the hash table. If a matching phrase $\gamma$ is found, then a replacement is made. Then $\gamma$ is combined with its preceding symbol, the one that preceded $\alpha_{i-1}$. This continues until a replacement is no longer possible. Then, the next symbol in $S_1$ is read and the process repeats. Phase 2a completes when the end of $S_1$ has been reached.

The memory requirements of phase 2a include the phrase hierarchy, without symbols expanded, and a buffer for the initial sequence. Since the replacement $\gamma$ is at least one
5.3 Identifying Old Symbol Pairs

Algorithm 5.1: Algorithm for RE-MERGE phase 2a.

```
input : \( P_1 \) and \( S_1 \)
output : \( P' \) and \( S' \)

1. foreach symbol in \( P_1 \) do
   2. insert symbol into hash table using its components

3. for \( i < |S_1| \) do
   4. replacement \( \leftarrow \) TRUE
   5. while replacement = TRUE do
      6. \( \gamma \leftarrow \) searchHashTable(\( \alpha_{i-1}, \alpha_i \))
      7. if \( \gamma \) found then
         8. \( \alpha_{i-1} \leftarrow \gamma \)
         9. \( i \leftarrow i - 1 \)
      10. else
         11. replacement = FALSE
      12. \( P' \leftarrow P_1; S' \leftarrow S_1 \)
   13. return (\( P', S' \))
```

generation higher than \( \alpha \) and \( \beta \), the maximum number of consecutive replacements cannot
be more than the maximum symbol generation in the phrase hierarchy. Therefore, a buffer
whose size is equal to at least the maximum symbol generation is sufficient. As experiments
later show, applying character-based RE-PAIR on blocks from 508 MiB of text produces a
phrase hierarchy whose highest symbol generation is 31.

With respect to execution time, phase 2a performs two main operations. First, all of
the symbols in the phrase hierarchy are inserted into the hash table. Second, every symbol
is combined with its preceding symbol for searching in the hash table. If we assume that
the size of the hash table is selected so that the length of the chains are about 1, then
accessing it during creation or searching requires constant time. So, the first operation can
be performed in \( O(|P_1|) \) time. The second task requires \( |S_1| \) hash table look-ups when no
replacements are made. If the number of replacements made is taken into account, then
note that the most substitutions an input symbol can participate in cannot be more than
the highest symbol generation. That is, if the highest symbol generation is \( g \), then the
second task runs in \( O(g|S_1|) \). After phase 2a completes, the number of searches into the
Block Merging for Re-Pair

hash table can be given as $|S_1| + (|S_1| - |S'|) \leq g|S_1|$. A single replacement during phase 2a reduces the length of $|S_1|$ by one symbol and adds one search operation into the hash table. As the sequence usually has more symbols than the phrase hierarchy, the execution time of phase 2a is $O(g|S_1|)$.

This phase of Re-MERGE is able to locate all pairs of symbols in the sequence that can be replaced. Phase 2a is fast and memory efficient, as demonstrated in experiments later in this chapter.

Effective Sequence Reduction

While phase 2a locates all pairs of symbols in the sequence produced by phase 1, the choices made may not be the best. A replacement now may prevent an even better replacement later that ends up reducing the overall length of the sequence. As an alternative to phase 2a, phase 2b optimally re-parses the sequence using the merged phrase hierarchy as a reference point. Phase 2b should not be used in conjunction with phase 2a, as doing so achieves no benefit.

Phase 2b is based on work by Katajainen and Raita [1989] and Klein [1997], and involves the re-parsing, or recompression, of the original message using phrases from a dictionary. Like the previous investigations in recompression, phase 2b converts the problem of compression to that of determining the single source shortest path in a directed acyclic graph (DAG). Unlike the earlier work, phase 2b is tailored specifically for Re-PAIR, in that it exploits the structure in the phrase hierarchy and sequence.

Phase 2b constructs a graph $G$ with a set of vertices $V$ connected by a set of edges $E$ denoted as $G(V,E)$. Each vertex $v_i \in V$ represents the $i$th primitive in the original message. An additional vertex is added to $V$ as a sink node. For a message of $n$ primitives, $V$ has $n + 1$ vertices.

Edges are attached to the vertices based on the strings in the phrase hierarchy. Vertices are examined in the order dictated by the original message, with the sink node being last. A directed edge $e_j \in E$ is drawn from $v_i$ to $v_{i+k}$, for $k > 0$, if a symbol exists in the phrase hierarchy which, when expanded, coincides with the $k$ primitives starting from $v_i$ up to,
5.3 Identifying Old Symbol Pairs

<table>
<thead>
<tr>
<th>Phrase hierarchy index</th>
<th>Expanded phrase</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>o</td>
</tr>
<tr>
<td>4</td>
<td>od</td>
</tr>
<tr>
<td>5</td>
<td>oo</td>
</tr>
<tr>
<td>2</td>
<td>w</td>
</tr>
<tr>
<td>6</td>
<td>wo</td>
</tr>
<tr>
<td>7</td>
<td>woo</td>
</tr>
<tr>
<td>8</td>
<td>wood</td>
</tr>
<tr>
<td>9</td>
<td>woody</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
</tr>
</tbody>
</table>

Table 5.2: The final phrase hierarchy of Figure 5.3, sorted in alphabetical order. The indexes in the first column refer to the positions of the symbols in the unsorted phrase hierarchy.

and including, \( v_{n+k-1} \). Since every primitive in the message exists in the phrase hierarchy, every vertex has at least one edge leading to the next vertex, and the graph is guaranteed to be connected. The vertex representing the last primitive has an edge leading to the sink node \( v_{n+1} \). Every edge has a weight of one.

After all possible edges are added to \( E \), a new sequence \( S' \) is created by finding the shortest path from \( v_1 \) to the sink node \( v_{n+1} \), thereby minimising the total number of edges traversed. Because the edges have unit cost, the shortest path can be determined through breadth-first search [Cormen et al., 2001], and represents the optimal parsing of the message using the phrase hierarchy, if optimality is measured by the number of edges spanned.

Figure 5.5 shows a graph for the merged phrase hierarchy and the beginning of the decoded message from Figure 5.3. That phrase hierarchy has been reproduced in Table 5.2, sorted in alphabetical order. In Figure 5.5, the optimal parsing for the first 7 primitives consists of the two symbols 7 and 8, obtained by traversing the dashed edges.

If the phrase hierarchy is sorted, edges can be selected for \( E \) through binary search. For example, suppose the vertex \( v_1 \) in Figure 5.5 is currently being processed. Let the string starting from this vertex, “woowoodw...”, be denoted as \( \tau_1 \). A binary search for \( \tau_1 \) in the phrase hierarchy would return “woo” as the longest match. By reducing the
Block Merging for Re-Pair

Figure 5.5: A graph representation of part of the message from the example of Figure 5.3. The symbols in the final phrase hierarchy of that figure allow edges to be added. The shortest path through these 8 vertices is shown as dashed lines.

number of significant primitives in \( \tau \), other edges from \( v_1 \) can be found, until the shortest edge of length 1 has been located. This process must be repeated for every vertex in the graph.

The cost of finding just the longest match for every vertex in a graph derived from a message of \( n \) primitives is \( O(n \log_2 |P_1|) \) string comparisons. A slight improvement is possible when an index into the phrase hierarchy is set up for the \( |\Sigma| \) primitives to constrain the binary search. For example, the longest match for “woowoodw...” must exist between “w” and “x”, assuming both exist as primitives in the phrase hierarchy. In the worse case, if every symbol begins with the same primitive, then such a data structure would neither improve nor significantly worsen the execution time. While not implemented for the experiments later, the size of the index could increase by indexing on more than one primitive.

Phase 2b applies binary search with this index structure for primitives, while also exploiting other characteristics of the phrase hierarchy, and of the symbols in the initial sequence. There are four main steps to Re-MERGE phase 2b, which are described in the next few paragraphs. The first step adds links to each symbol in the phrase hierarchy. The second step expands the initial sequence into primitives to form \( V \). Then, edges are added to \( E \) in the third step. Finally, \( S' \) is output by determining the shortest path through the vertices.
5.3 Identifying Old Symbol Pairs

Additional links for the phrase hierarchy

Phase 2b assumes that each symbol in the phrase hierarchy has been expanded into primitive strings and then sorted on these strings. Three types of links are added to each symbol in the phrase hierarchy in preparation for phase 2b. The three link types for each symbol lead to a primary prefix symbol, a secondary prefix symbol, and a list of extended symbols.

A symbol $\beta$ is a prefix of another symbol $\alpha$, if $|\beta| \leq |\alpha|$ and the expanded string of $\beta$ coincides with the first $|\beta|$ primitives of $\alpha$. A symbol $\beta$ is the primary prefix symbol of $\alpha$ if $\beta$ is the longest prefix symbol in the phrase hierarchy for $\alpha$. A symbol $\beta$ is the secondary prefix symbol of $\alpha$ if $\beta$ is the longest prefix of $\alpha$, starting from $\alpha$'s second primitive. The primary and secondary prefix symbols of a primitive are the empty symbol, $\lambda$. For example, in the phrase hierarchy of Table 5.2, the symbol “wood” has “woo” and “oo” as its primary and secondary prefix symbols, respectively. As no two symbols in the phrase hierarchy expand to the same string after phase 1, every symbol is guaranteed to have exactly one symbol that serves either role.

The symbol $\beta$ is an extended symbol of $\alpha$, if two conditions hold. First, $\alpha$ must be the primary prefix symbol of $\beta$. Second, for some primitive $\delta$, no symbol which is lexically smaller than $\beta$ exists which has the string $\alpha\delta$ as a prefix, where the juxtaposition of symbols represents string concatenation. According to this definition, the string $\alpha\delta$ does not have to exist in the phrase hierarchy, but this is possible if $\beta$ is longer than $\alpha$ by one primitive. For example, in Table 5.2, “wood” is an extended symbol of “woo”, where $\delta = \text{"d"}$.

A symbol has a list of zero to at most $|\Sigma|$ extended symbols. An extended symbol is recorded as a pair, of the form $(\delta, \beta)$, and ordered in the list according to $\delta$. The pair $(\delta, \beta)$ states that, for the current symbol $\alpha$, the primitive $\delta$ can be appended to it, and the lexically smallest symbol in the sorted phrase hierarchy with this string as a prefix is $\beta$. When the list of extended symbols is non-empty, then a special pair of the form $(\epsilon, \beta)$ is added to the end. This pair, called an $\epsilon$-pair, indicates that in the lexically sorted phrase hierarchy, no symbol has $\alpha$ as a primary prefix symbol at or after $\beta$. Also, the symbol
Block Merging for Re-Pair

<table>
<thead>
<tr>
<th>Phrase hierarchy index</th>
<th>Expanded phrase</th>
<th>Primary prefix symbol</th>
<th>Set of extended symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>wo</td>
<td>847</td>
<td>[ (“o”, 1000), (“r”, 1009), (ε, 1121) ]</td>
</tr>
<tr>
<td>1000</td>
<td>wood</td>
<td>999</td>
<td>[ (“b”, 1001), (“c”, 1002), (ε, 1008) ]</td>
</tr>
<tr>
<td>1001</td>
<td>woodblock</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>woodcarving</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1003</td>
<td>woodchip</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>woodchuck</td>
<td>1000</td>
<td>[ (“s”, 1005), (ε, 1006) ]</td>
</tr>
<tr>
<td>1005</td>
<td>woodchucks</td>
<td>1004</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>woodcraft</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1007</td>
<td>woodcut</td>
<td>1000</td>
<td>[ (“t”, 1008), (ε, 1009) ]</td>
</tr>
<tr>
<td>1008</td>
<td>woodcutter</td>
<td>1007</td>
<td></td>
</tr>
<tr>
<td>1009</td>
<td>work</td>
<td>999</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Binary search on the phrase hierarchy using extended symbols. The numbering for the index (first column) is referred to by the third and fourth columns. In the second column, every symbol has been expanded into primitives, and the entire phrase hierarchy has been sorted on this column.

immediately before \( \beta \) must have \( \alpha \) as a primary prefix symbol.

Table 5.3 further clarifies the definition of extended symbols by demonstrating how they are used. The table shows a section of a sorted phrase hierarchy from the symbol “wo” to “work”, with the symbols expanded into strings. Also shown are every symbol’s primary prefix symbol and list of extended symbols. Suppose \( \tau_1 \) = “woodcutters” and, through methods described below, the current prefix for \( \tau_1 \) is the string “wood”, but whether a longer one exists is unknown. Since the first four primitives of “wood” match with \( \tau_1 \) and the next primitive in \( \tau_1 \) is available, the longest match must exist between “wood” and the lexically largest symbol that has “wood” as a prefix symbol and is followed by a “c”. These two boundaries can be established through two consecutive extended symbol pairs for the symbol “wood”. While a binary search within the search space allows a longer prefix to be found for \( \tau_1 \), locating the longest prefix requires this process to be repeated. Rather than applying binary search on the entire phrase hierarchy, extended symbols permit sections of the phrase hierarchy to be isolated based on the primitives that have already been compared.
5.3 Identifying Old Symbol Pairs

The steps performed for the example of Table 5.3 is as follows. The extended symbols of “wood” indicate that a “c” can be appended to it. The first such symbol is symbol 1002, “woodcarving”. The last symbol that has “wood” as a primary prefix symbol and “c” as the fifth primitive is “woodcut”, which is one symbol before the ε-pair to “woodcutter”. A binary search between these two symbols, shown shaded in Table 5.3, is performed next. Within this interval, all of these strings have the same first five primitives. However, not all of them share the same primary prefix symbol. The longest match within this search interval is “woodcut”, but it is still not the longest match for $\tau_1$. An attempt to extend “woodcut” using the next primitive, “t” is made. It succeeds and a search interval of one symbol (1008) is found. This symbol cannot be extended further, so the longest match for $\tau_1 = \text{"woodcutters}”$ is “woodcutter”.

As an example of primary prefix symbols, secondary prefix symbols, and sets of extended symbols, Table 5.4 shows these links for the phrase hierarchy of Table 5.2. The first two columns of the figure show the phrase hierarchy indexes and expanded strings. The next two columns show the primary and secondary prefix symbols. In the last column, the lists of extended symbols are shown for symbols that are primary prefix symbols of another symbol. In case an ε-pair points outside of the phrase hierarchy, an extra symbol has been added at position 10 of the phrase hierarchy.

Phase 2b requires the phrase hierarchy to be sorted by strings of expanded symbols. Then, symbols are linked through a series of passes. Primary prefix symbols are assigned first by comparing each symbol with its immediately preceding symbols. When searching, the first prefix symbol located serves as the primary prefix symbol, due to the sorted order. While it is possible for a symbol to be compared with every symbol before it, in practice, a match is usually found after a few strings have been checked. A second pass is done to attach symbols to the lists of extended symbols, using the primary prefix symbol links. Each symbol adds or updates an extended symbol pair to the list associated with its primary prefix symbol. A third pass makes use of the other two types of links to assign the secondary prefix symbols. With the phrase hierarchy sorted according to expanded symbols, consecutive symbols are expected to have the same, or similar, secondary prefix
### Block Merging for Re-Pair

<table>
<thead>
<tr>
<th>Phrase hierarchy index</th>
<th>Expanded phrase</th>
<th>Primary prefix symbol</th>
<th>Secondary prefix symbol</th>
<th>Set of extended symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>d</td>
<td>λ</td>
<td>λ</td>
<td>[ (&quot;d&quot;,&quot;4&quot;), (&quot;o&quot;,&quot;5&quot;), (y(\varepsilon,2)) ]</td>
</tr>
<tr>
<td>1</td>
<td>o</td>
<td>λ</td>
<td>λ</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>od</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>oo</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>w</td>
<td>λ</td>
<td>λ</td>
<td>[ (&quot;o&quot;,&quot;6&quot;), ((\varepsilon,7)) ]</td>
</tr>
<tr>
<td>6</td>
<td>wo</td>
<td>2</td>
<td>1</td>
<td>[ (&quot;o&quot;,&quot;7&quot;), ((\varepsilon,8)) ]</td>
</tr>
<tr>
<td>7</td>
<td>woo</td>
<td>6</td>
<td>5</td>
<td>[ (&quot;d&quot;,&quot;8&quot;), ((\varepsilon,9)) ]</td>
</tr>
<tr>
<td>8</td>
<td>wood</td>
<td>7</td>
<td>5</td>
<td>[ (&quot;y&quot;,&quot;9), ((\varepsilon,3)) ]</td>
</tr>
<tr>
<td>9</td>
<td>woody</td>
<td>8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td>λ</td>
<td>λ</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>(End of phrase hierarchy)</td>
</tr>
</tbody>
</table>

Table 5.4: The additional symbol links required for the phrase hierarchy before starting phase 2b. Primary and secondary prefix symbols are shown in the third and fourth columns, after the expanded symbols. These symbols do not exist for primitives, and are indicated by the empty symbol, \(\lambda\). In the last column, sets of extended symbols are either empty, or made up of at least two pairs, with the last pair being of the form \((\varepsilon,\alpha)\). Note that the pair \((\varepsilon,7)\) of symbol 2 points to symbol 7 and not 10. While symbols 7 to 9 start with "w", none of them have a primary prefix symbol of 2.

This observation assists in the creation of this third link, in the absence of re-sorting the phrase hierarchy from each string’s second primitive.

The three types of links add \(6|P_1|\) words of memory to the phrase hierarchy of phase 1. Every symbol in the phrase hierarchy has exactly one primary prefix symbol and one secondary prefix symbol, each stored in one word of memory. An extended symbol pair requires two words of memory, and there cannot be more than \(|P_1|\) of them. As proof of this claim, note that one of the conditions for \(\alpha\) to be an extended symbol of \(\beta\) is that \(\beta\) is the primary prefix of \(\alpha\). Since \(\alpha\) has an unique primary prefix symbol, each symbol in the phrase hierarchy participates as an extended symbol only once. Finally, for each non-empty set of extended symbols, there must be an extra \(\varepsilon\)-pair to end the list of pairs. Each of these require two words of memory.
5.3 Identifying Old Symbol Pairs

![Diagram showing S, primitives, and initial search position]

Figure 5.6: Example decoding of $S_1$ into vertices, represented as an array. Each vertex keeps track of the primitive and an initial starting position.

Decoding the sequence

After the symbol links have been established for the phrase hierarchy, the next step is to expand the symbols in $S_1$ into $\tau$. For a message of $n$ primitives, this operation requires memory proportional to $n$. Alternatively, $S_1$ can be decoded a symbol at a time so that the length of $\tau$ is at least equal to the length of the longest symbol in the phrase hierarchy, $\alpha_L$.

While expanding the sequence into $\tau$, the original symbols in $S_1$ are maintained to improve search times later. Each vertex is augmented with an initial search position. The initial search position at vertex $v_i$ is a symbol which is guaranteed to be the lower bound in a binary search for the longest prefix for $\tau_i$. For example, if $\tau_1$ is the string “woowoodw...”, then “w” is one possibility for $v_1$. However, a longer initial search position is possible if $S_1$ is taken into account. When a symbol is decoded into primitives for $\tau_i$, the symbol may not represent the longest match, but it may still be a better match than a single primitive. This is illustrated in Figure 5.6 using the final phrase hierarchy and sequence from Figure 5.3. The first two symbols of $S_1$ are 7 and 8. The primitives at the beginning of the expanded symbols acquire these 2 symbols as initial search positions, as shown in the figure.

The remaining initial search positions are initialised with symbols representing single primitives. However, each are later replaced by the secondary prefix symbol of the preceding vertex, if it is longer. For example, in Figure 5.6, after the longest edge representing “woo” is added to $v_1$, its secondary prefix symbol is “oo”. This symbol replaces the initial search position at $v_2$, which is the primitive “o” (shown shaded).
Figure 5.6 also demonstrates how only the minimum number of symbols need to be decoded for \( \tau_i \). If the current vertex being processed is \( v_i \), then since \( |\alpha_L| = 5 \) for Table 5.4, only two symbols need to be decoded to create \( \tau_i \) with a length of 7 primitives. Additional symbols are decoded as phase 2b progresses through \( \tau \).

**Adding edges to the graph**

After enough of the sequence has been decoded so that \( |\tau_i| \geq |\alpha_L| \), edges are added to \( v_i \), starting with the longest one. The longest edge from \( v_i \) is also the primary prefix of \( \tau_i \).

Let the symbol \( \beta \) represent the best prefix symbol so far and \( \delta \) be the primitive at position \( |\beta| + 1 \) in \( \tau_i \). The symbol \( \beta \) is initialised with the initial search position of \( v_i \), and is concatenated with \( \delta \) for searching in the extended symbols of \( \beta \). If the search is unsuccessful, then \( \beta \) is the longest match. But if the search is successful, two consecutive symbol pairs are retrieved, where the first one is the pair \((\delta, \alpha_i)\). These two pairs, \( \alpha_{\text{left}} \) and \( \alpha_{\text{right}} \), form an area within the phrase hierarchy for binary search. If the binary search fails, the current version of \( \beta \) is the longest primary prefix. If the search succeeds, then a longer string is returned. The process repeats with this longer string by first extending it with its next primitive in \( \tau_i \).

Once the longest edge has been determined and added, all shorter edges are located by repeatedly following prefix symbol links until the empty symbol \((\lambda)\) is reached.

**Determining the shortest path**

The last step of phase 2b requires the calculation of the shortest path through the graph. Earlier, the number of symbols in the sequence that have to be decoded was reduced by noticing that only the first \( |\alpha_L| \) primitives in \( \tau_i \) are significant. Another observation divides the expanded sequence into segments in order to prevent the entire message from being kept in memory.

A *segment* is defined as being a sequence of vertices which begins and ends with *cut vertices* (also called *choke points*). A cut vertex is a vertex which has edges starting or ending at it, but no edges that bypass it. The first vertex \( v_1 \) and the sink node \( v_{n+1} \) are
both examples of cut vertices. Other cut vertices also exist in most sequences and can be taken advantage of.

Since no edges bypass a cut vertex, the shortest path through the set of vertices must go through that vertex. Therefore, if the graph $G$ is processed in segments, no loss in the optimal parsing of the message occurs. The shortest path can be determined a segment at a time, rather needing to have all of $V$ in memory.

Figure 5.7 demonstrates how the first few primitives of Figure 5.6 are handled using the phrase hierarchy and symbol links of Table 5.4. Three pointers to the vertices are maintained: $min$, $max$, and $i$. The vertex at position $i$ is currently being examined. The $min$ pointer represents the beginning of the current segment, while $max$ points to the vertex that is reached by the longest edge. Whenever the vertex currently being examined meets $max$, a cut vertex has been found, as in Figure 5.7(d).

An examination of Figure 5.7 reveals an alternative method of performing phase 2b, which has not been attempted. Currently, at each step, the vertex at position $i$ is examined to determine all of the edges that start from $i$. Another way of performing phase 2b is to determine all edges that end at $i$, instead. This method of conducting phase 2b has the potential of being more memory efficient. As edges are added, others may be removed if a better way of reaching the current vertex $i$ has been found. At least one change to the current phase 2b algorithm is required in order to use this proposed version. While phase 2b requires the phrases in the hierarchy to be sorted in the forward direction, this proposed version would require all of the strings to be reversed, and then sorted.

The segment from $min$ to $max$ is processed by finding the shortest path through the segment. The shortest path through a segment is determined by using breadth-first search, which has a running time of $O(|V| + |E|)$ [Cormen et al., 2001]. After the shortest path has been found, the symbols that represent that path are added to the final sequence $S'$. At this point, the memory taken up by the segment is released before starting on the next segment with $i$ and $min$ starting from $max$. 
Figure 5.7: Examining vertices to locate the next cut vertex. The pointer $i$ indicates the vertex that is being considered. When $i$ coincides with the pointer $max$, a cut vertex has been found, and the next segment is from $min$ to $max$. Note that when $|r_i|$ is less than the length of the longest symbol, more symbols need to be decoded, as in the lower two figures.

**Analysing Phase 2b**

The algorithm for phase 2b is shown in Algorithm 5.2. This algorithm excludes the sorting of the phrase hierarchy by expanding symbols and the addition of symbol links. Algorithm 5.2 requires $P_1, S_1$, and the sorted phrase hierarchy $P_s$ as input. At the end, $P'$ and $S'$ are produced by the algorithm.

The main loop, starting at line 3, continues until the initial sequence $S_1$ has been exhausted. The sequence is decoded a symbol at a time into primitives on line 4 to
5.3 Identifying Old Symbol Pairs

**Algorithm 5.2:** Algorithm for RE-MERGE phase 2b.

<table>
<thead>
<tr>
<th>input</th>
<th>( \mathcal{P}_1, \mathcal{P}_s, ) and ( \mathcal{S}_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>( \mathcal{P}' ) and ( \mathcal{S}' )</td>
</tr>
</tbody>
</table>

1. \( \alpha_L \leftarrow \) longest symbol in \( \mathcal{P}_1 \)
2. \( \min \leftarrow 0; i \leftarrow 1 \)
3. while not end of \( \mathcal{S}_1 \) do
   4. decode and expand next symbol in \( \mathcal{S}_1 \) and append the expansion to \( \tau_i \)
   5. while \( |\tau_i| > |\alpha_L| \) do
      6. \( \beta \leftarrow \) initial search position at \( v_i \)
      7. while longest match not found do
         8. \( \delta \leftarrow \) primitive at \( v_{i+|\beta|} \)
         9. \( (\alpha_{\text{left}}, \alpha_{\text{right}}) \leftarrow \text{findExtendSymbol}(\beta, \delta) \)
         10. if \( \alpha_{\text{left}} = \lambda \) then break
         11. \( \text{depth} \leftarrow |\beta| \)
         12. \( \beta_{\text{temp}} \leftarrow \text{binarySearch}(\tau_i, \mathcal{P}_s, \alpha_{\text{left}}, \alpha_{\text{right}}, \text{depth}) \)
         13. if \( \beta_{\text{temp}} = \lambda \) then break
         14. \( \beta \leftarrow \beta_{\text{temp}} \)
      15. while \( \beta \neq \lambda \) do
         16. addEdge(\( \beta, v_i \))
         17. update \( \text{max} \)
         18. update initial search position of \( v_{i+1} \)
      19. \( i \leftarrow i + 1 \)
   20. if \( i = \text{max} \) then
      21. \( \text{path} \leftarrow \text{findShortestPath}(v_{\min}, v_{\max}) \)
      22. append \( \text{path} \) to \( \mathcal{S}' \)
      23. \( \min \leftarrow \text{max} \)
   24. \( \mathcal{P}' \leftarrow \mathcal{P}_1 \)
25. return \( (\mathcal{P}', \mathcal{S}') \)

form the string \( \tau_i \), where \( i \) is the position of the primitive being considered. Decoding temporarily stops on line 5 whenever \( \tau_i \) is at least as long as the longest symbol in the phrase hierarchy, \( \alpha_L \). Then, edges are added to \( v_i \) through repeated iterations of symbol extensions and binary search (lines 5 to 18). Before proceeding with vertex \( v_{i+1} \), its initial search position is updated with the secondary prefix symbol of \( v_i \) on line 19. Next, a check is performed on line 21 to determine whether \( i \) is equal to \( \text{max} \). If so, then the end of a segment has been found, and the shortest path through that segment is calculated. The symbols that represent this path is appended to the output sequence, \( \mathcal{S}' \). After all of \( \mathcal{S}_1 \)
has been processed, $P'$ is simply a copy of $P_1$, since phase 2b does not change the phrase hierarchy.

Sorting the phrase hierarchy requires $O(|P'||\log_2 |P'|)$ string comparisons, while the addition of the three types of symbol links is performed via a series of linear passes through the phrase hierarchy, as mentioned earlier.

The time needed for the remainder of phase 2b, as shown in Algorithm 5.2, can be calculated by dividing the algorithm into three main operations. If an assumption is made that only two cut vertices are found throughout the graph ($v_1$ and $v_{n+1}$), then these three tasks can be considered as if they exist in isolation. Finding the longest edge for all vertices requires a binary search at every vertex in the graph $G$. So, this runs in $O(n|P'| \log_2 |P'|)$ time. This analysis assumes that secondary prefix symbols and extended symbols fail to improve recompression efficiency, without factoring in the overhead cost of searching through lists of extended symbols. Adding shorter edges to a vertex once the longest one has been found requires constant time due to the primary prefix symbol links. Because of this, the second task runs in $O(n)$ time. Finally, as mentioned earlier, finding the shortest path through the segment takes $O(|V| + |E|)$ time, where $|V|$ and $|E|$ are the number of vertices and edges in the graph. Alternatively, this time can be re-written as $O(n + |E|)$. Summing all of these parts of phase 2b results in a running time whose dominant term is $O(n|P'| \log_2 (|P'|))$.

The three extra links contribute an additional $6|P'|$ words of memory to the phrase hierarchy. So that this phase runs efficiently, symbols may be expanded completely or with a recently-used buffer, as described for Des-Pair in Section 3.1 on page 48. If the entire message is treated as a single segment, then the expanded sequence requires $2n + |E|$ words of memory. Two words of memory are required for the primitive and initial search position of each vertex. Furthermore, a word of memory is required for each edge added to the graph. Since the message occupies more space than the phrase hierarchy in practice, phase 2b operates in space linear to the length of the message.

However, this analysis of the time and space of phase 2b assumes that no internal cut vertices are found, and that secondary prefix symbols and extended symbols are unhelpful.
5.3 Identifying Old Symbol Pairs

Later, experiments demonstrate that this is not the case. Both the running time and space requirements of phase 2b have been overestimated in this analysis. While most time is still spent adding edges to the graph, the frequency with which cut vertices appear ensure that most of the space is attributed to the phrase hierarchy, and not the message. This is shown in the experiments later in this chapter.

Comparison with Related Work

Phase 2b of RE-MERGE closely follows the work by Katajainen and Raita [1989] and Klein [1997]. Katajainen and Raita provides a description of recompression that is applicable to any system, regardless of how the dictionary is created. The extended trie structure of Aho and Corasick [1975] is built in order to search for the longest match. While phase 2b aims to minimise the length of the sequence, the system by Katajainen and Raita presumes that codewords are already assigned to each dictionary entry and edge. As a result, the compression ratio of the sequence is minimised rather than the sequence length.

This is in contrast to RE-MERGE, which follows the same design paradigm as RE-PAIR of separating the model from the coder, a choice that was necessary because RE-PAIR replaces pairs and builds the phrase hierarchy simultaneously. However, with the phrase hierarchy fixed in phase 2b, assigning weights to edges corresponding to codeword lengths could lead to small gains in compression effectiveness. Aspects of the implementation by Katajainen and Raita which are similar to phase 2b include the use of cut vertices, and a top-level index for primitives.

The recompression technique of Klein [1997] is more specific and is described in the context of the LZ77 family of compression algorithms. It introduces a method of pruning which reduces the number of vertices and edges maintained without any loss in recompression effectiveness. If memory limitations exists for phase 2b, then these pruning techniques can be adopted. The experiments described below do not make use of them.

Phase 2b deviates from previous work by taking advantage of the prior knowledge about the phrase hierarchy and applying binary search to it. Katajainen and Raita created a trie data structure because their algorithm was general and no presumptions could be
Block Merging for Re-Pair

made about the dictionary. In our case, it is guaranteed that every symbol in the phrase hierarchy of phase 2b can be broken down into primitives, which all exist in the dictionary. The relationships between symbols permit the definitions of the three types of symbol links. Also, the phrase hierarchy is static throughout phase 2b, an assumption not available to Klein. More importantly, the RE-PAIR phrase hierarchy creates many opportunities for cut vertices during compression, and complex pruning techniques are not necessary.

The segmentation of the message through cut vertices ensures that the memory usage of phase 2b is limited by the phrase hierarchy, and not the sequence. One problem with the algorithm is whether the additional symbol links yield enough benefit for binary searching to justify them. This question is addressed later during experiments with phase 2b when a comparison with a more simplistic method is made.

5.4 Identifying New Symbol Pairs

Regardless of whether phase 2a or phase 2b is applied, the resulting sequence contains no pair of symbols with a replacement in the phrase hierarchy. But it is possible that there are pairs of symbols in the new sequence which are not recorded in the phrase hierarchy. For example, if \( \alpha \beta \) exists only once in each of blocks 1 and 2, no replacement symbol would have been added to the phrase hierarchy by RE-PAIR or any other earlier phase of RE-MERGE. The purpose of phase 3 is to identify new symbol pairs that occur twice or more, and to replace them with a fresh symbol.

To achieve this purpose, phase 3 further reduces the length of the sequence. As with the second phase of RE-MERGE, a decrease in compression ratio cannot be guaranteed due to the separation of the coder from RE-PAIR. For each unique pair of symbols found which occur twice or more, an addition is made to the phrase hierarchy. Another intention of phase 3 is to improve the quality of the phrase hierarchy, by creating longer phrases.

Several methods of performing phase 3 on the input sequence \( S_1 \) are conceivable. The first method simply applies RE-PAIR to \( S_1 \). But even after the application of phases 1 and 2, the length of \( S_1 \) going into phase 3 may still be too long, and impossible to process as a single RE-PAIR block. Alternatively, \( S_1 \) can first be divided into blocks. However, it
5.4 Identifying New Symbol Pairs

is possible that no amount of iterations on $S_1$ will reduce the sequence to a length that can ultimately be processed entirely within memory by RE-PAIR. Instead of these two, a more direct approach is taken by phase 3.

As was the case in the other phases in RE-MERGE, the amount of memory occupied by the chosen method for phase 3 depends on the size of the phrase hierarchy, as opposed to the length of the sequence. The phrase hierarchy is retained in memory while four complete passes are performed on $S_1$. In the first pass, pairs of symbols are maintained in a probabilistic filter implemented as a bit vector, $B$. For every pair of symbols in the initial sequence, a set of independent hash functions are applied, identifying a set of bits in $B$, which are all turned on. Subsequent occurrences of the same pair are identified when every bit in the set are already on. In this case, the symbol pair is added to a hash table, $H$. Before completing the first pass, the bit vector’s memory is released.

There is no attempt in $B$ to resolve collisions, and each bit is either on or off. So, even with the best set of independent hash functions, it is possible for false hits to occur in $B$ that result in a symbol pair being added to $H$ after a single occurrence in $S_1$. A second pass through $S_1$ confirms which of the entries in $H$ are valid replacements. For each entry, a counter indicates whether the corresponding pair of symbols has been seen only once, or seen more than once. Like RE-PAIR, care must be taken to prevent inflating the count for overlapping pairs; a problem which is easily solved by keeping track of the previous pair of symbols at all times. Once the second pass completes, the entries in the hash table are examined so that symbol pairs that have been seen only once are culled.

The third pass replaces symbol pairs in $S_1$ according to $H$, and adds symbols to the phrase hierarchy as each new replacement is performed. Unlike RE-PAIR, replacements are performed in a strictly left-to-right manner, starting from the beginning of the sequence, regardless of pair frequency. Since most lower generation phrases have already been found by RE-PAIR, the majority of the pairs located by phase 3 are expected to be infrequent, and associated with a high generation. Only a small impact on the overall quality of the phrases is expected, even though phase 3 does not pair symbols recursively. But if recursive pairing is necessary, then multiple iterations of phase 3 over the sequence can be
made, similar to the compression system RAY by Cannane and Williams [2001].

Finally, a fourth pass over the sequence is needed because of the requirements of phrase hierarchy encoding. As the new symbols may be introduced into any generation, except for the generation of primitives, the phrase hierarchy needs to be built one more time. Rebuilding requires the entire phrase hierarchy to be sorted by generation, each symbol assigned a chiastic slide value, and then each generation interpolative coded. After the phrase hierarchy is encoded, the fourth pass renumbers the sequence \( S_1 \) to ensure that the symbols refer to the final phrase hierarchy \( \mathcal{P}' \).

Phase 3 operates in time linear in the length of the initial sequence \( S_1 \) excluding the decoding and encoding of the phrase hierarchy. To be precise, the first three passes scan a sequence of length \( |S_1| \), while the fourth pass scans a sequence of length \( |S'| < |S_1| \). In addition, the second pass requires a scan over the hash table to remove pairs that were falsely identified.

The separation of phase 3 into passes reduces the amount of memory used. For example, if the bit vector \( B \) was not used, then the hash table \( H \) could occupy as much memory as there are symbols in \( S_1 \). Instead, the memory usage of phase 3 is dominated by the size of the phrase hierarchy, which does not require symbols to be expanded or inter-linked, unlike phases 1 and 2b. The sequence is read directly from disk for each pass of phase 3 so that it is unnecessary to keep any part of it in memory.

Another factor that affects the memory usage of phase 3 has yet to be clarified. The number of entries in \( H \) is at least equal to the number of replacements made, but could be higher depending on the number of false duplicates that arise during the first pass. The change of a false duplicate depends on the size of the bit vector \( B \) and the number of independent hash functions. Let the size of the bit vector be \( |B| \) and the number of independent hash functions be \( k \). The highest chance for a false duplicate occurs when all of \( S_1 \) has been processed. The probability of any particular bit in \( B \) still being off after \( k|S_1| \) bits have been randomly turned on is \( (1 - 1/|B|)^{k|S_1|} \). Thus, the probability that
5.4 Identifying New Symbol Pairs

| $(|S_1|)$ | $(|S_2|)$ | $(|S_3|)$ | $(|S_4|)$ |
|---|---|---|---|
| 3 | 5 | 12 | 18 |
| 6 | 10 | 25 | 38 |
| 12 | 15 | 37 | 56 |

Figure 5.8: Percentage calculations for the chance of obtaining a false duplicate using Equation 5.1. On the left, $(|S_1|) = 61,734$ and on the right, $(|S_2|) = 61,734,292$. The number of hash functions (on the left side of each table) varies from 1 to 7, while the size of $B$, measured in bits along the top, varies from 8 bits per symbol up to 32 bits per symbol.

A random selection of $k$ bits in $B$ are all on, is given by

$$\left(1 - \left(1 - \frac{1}{|B|}\right)^{|S_1|}\right)^k$$

Equation 5.1 assumes that the first pass through $S_1$ has completed. With the sparser distribution of bits turned on in $B$ while the sequence is scanned, this equation represents an upper bound on the probability of obtaining a false duplicate overall.

Figure 5.8 substitutes various values for $|B|$, $k$, and $|S_1|$ to provide actual probabilities, represented as percentages. The size of the bit vector depends on the length of the initial sequence. Also, because of efficiency in accessing individual bits, the various sizes of the bit vector were chosen so that they were byte-aligned. For example, in the tables, $(|S_1|)$ represents a bit vector with a size equal to 8 bits (or 1 byte) for each symbol in the initial sequence. The number of independent hash functions range from 1 to 7.

Percentage calculations with two sequence lengths are shown in Figure 5.8, with one being 1,000 times longer than the other. The sequence length of the table on the right is 61,734,292 symbols, a number taken from Table 5.6, later in this chapter. The negligible difference between the two tables show that Equation 5.1 is relatively insensitive to large changes in the sequence length, because of the way that $|B|$ is chosen as a function of $(|S_1|)$.

As expected, as $B$ increases in size relative to $(|S_1|)$, the chance of a false duplicate decreases. Moreover, as the first column in each table of Figure 5.8 show, as the number
of hash functions increase, the probabilities initially decrease, but ultimately increase again.

Based on these calculations, phase 3 has been implemented with three hash functions and a bit vector of size $8|S_1|$ bits, giving a false duplicate percentage of roughly 3.1%. This means that at the end of the scanning of the sequence $S_1$, the chance that a single additional pair of symbols would yield a false positive is about 3.1%. During the processing of $S_1$, the chance of a false positive is less. The number of slots in the hash table $H$ can be either fixed, or chosen based on the length of $S_1$.

### 5.5 Changes for Punctuation-Aligned Re-Pair

In the previous sections, the three phases of RE-MERGE have been discussed in the context of character-based Re-Pair. The reason for this presentation was to assist in the understanding of the examples throughout the discussion. However, as described in Chapter 4, word-based or punctuation-aligned Re-Pair are preferred for the purpose of phrase browsing. No changes to RE-MERGE are required for word-based Re-Pair. However, punctuation-aligned Re-Pair makes use of punctuation flags, which have to be handled appropriately.

Recall from the previous chapter that the main difference between character-based Re-Pair and word-based Re-Pair, is that the set of primitives is larger. And the difference between word-based Re-Pair and punctuation-aligned Re-Pair is that symbols within the word sequence have been flagged. A flagged symbol indicates that, in the message, the word token is followed by one of the punctuation marks identified by punctuation-aligned Re-Pair.

While no significant changes are necessary for phase 1, minor changes are require for the remaining phases in order to accommodate punctuation flags. The rules governing punctuation-aligned Re-Pair are presented in Table 4.8 on page 93. Phase 2a should stop the recursive pairing of symbols if any of these rules have been violated. A rule violation occurs if the left component is flagged, but the right component is not. As with Re-Pair, this modification ensures that non-flagged tokens are paired with flagged tokens.
only if the flagged token is the right component. Likewise, recursive pairing should also cease if both components are flagged, but the symbol before the left component is not.

When punctuation-aligned RE-PAIR is combined with phase 2b of RE-MERGE, primitives which have been flagged need to be treated differently, as shown in Figure 5.9. In particular, certain edges in the directed graph constructed in phase 2b are disallowed, depending on the location of the flagged primitives. Flagged primitives are represented as minor cut vertices in $G$. Whether edges are allowed depends on the table of Figure 5.9(c). Figure 5.9(a) and Figure 5.9(b) together constitute an example for phase 2b with punctuation-aligned RE-PAIR.

Of the six rules of Figure 5.9(c), the two rules preceded with an asterisk (*) indicate cases when an edge is not allowed. The remaining four rules indicate situations when an edge is permitted. Which rule is applied depends on three conditions: the vertex type before the tail of the edge, the vertex type before the head of the edge, and whether a minor cut vertex has been bypassed by an edge. Note that for the two cases where edges are disallowed, the edge must have passed over a cut vertex. So an edge of length 1 that leads to the immediately adjacent vertex is never deleted, and $G$ is still guaranteed to be connected. The end result after applying these rules is similar to the constraints as described above for phase 2a.

An example assists in justifying these conditions. Figure 5.9(a) presents a message derived from the “Woodchuck” message used as an example in earlier chapters, with four punctuation marks added. Figure 5.9(b) contains two graphs. The top graph illustrates edges which correspond to the four rules which permit edges, while the bottom graph shows the two cases when edges are not permitted. The vertices, each representing a word token, have been coloured so that the four lightly shaded vertices are flagged primitives, and the darkly shaded vertices are sink nodes. Edges are labelled with letters, indicating rules from Figure 5.9(c). The primitives traversed by the edge are listed in the last column of Figure 5.9(c).

As the example shows, any sequence of primitives between minor cut vertices is permissible (edges A and D). If punctuation marks are seen as marking off English phrases,
Block Merging for Re-Pair

much wood, as a woodchuck would chuck, if a woodchuck, could chuck wood.

(a) Sample message

(b) Graphs with valid and invalid edges

<table>
<thead>
<tr>
<th>Graph edge</th>
<th>Vertex before tail</th>
<th>Vertex before head</th>
<th>Pass a minor cut vertex?</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>W</td>
<td>W</td>
<td>No</td>
<td>a woodchuck</td>
</tr>
<tr>
<td>B</td>
<td>W</td>
<td>P</td>
<td>Only once</td>
<td>woodchuck would chuck,</td>
</tr>
<tr>
<td>*</td>
<td>C</td>
<td>W</td>
<td>Otherwise ...</td>
<td>woodchuck would chuck, if</td>
</tr>
<tr>
<td>D</td>
<td>P</td>
<td>W</td>
<td>No</td>
<td>could chuck</td>
</tr>
<tr>
<td>E</td>
<td>P</td>
<td>P</td>
<td>Yes</td>
<td>as a woodchuck would chuck, if</td>
</tr>
<tr>
<td>*</td>
<td>F</td>
<td>P</td>
<td>Yes</td>
<td>as a woodchuck would chuck, if a</td>
</tr>
</tbody>
</table>

(c) All possible combinations of edges

Figure 5.9: The rules that dictate the types of edges that can be removed by phase 2b of Re-MERGE for punctuation-aligned Re-PAIR. Of the six rules at the bottom, only the two that are preceded by an asterisk (*) disallow edges. Each rule is illustrated through one of two example graphs (b) which correspond to the sample message at the top (a).

then permitted edges can be viewed as edges which do not break a phrase. For edge B, three primitives are combined, with only the final primitive immediately followed by a punctuation mark. Two phrases are combined by edge E, and neither of them break any surrounding phrases.

The changes to the third phase of Re-MERGE resembles that of phase 2a. Only pairs permitted by phase 2a are permitted by phase 3. These rules are applied during all passes
of phase 3, not just the replacement pass.

## 5.6 Experiments

Experiments were conducted with two documents which are larger than the 20 MiB considered in previous chapters. The first document is **WSJ508**, also used earlier in this chapter. The second document is **NEWS**, already mentioned in Chapter 2. The **NEWS** collection consists of 1,000 MiB of news articles from the Wall Street Journal (**WSJ**) and the Associated Press (**AP**) with SGML markup. The first 508 MiB of the **NEWS** collection is **WSJ508**.

The test machine for these experiments is again a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache, with all execution times averaged over three trials run on an otherwise idle machine. Table 5.5 presents the results of using the **GZIP**, **BZIP2**, and **PPMD** systems on these two documents, running on the same test machine. These results are from Table 2.5 (page 41) and Table 2.6 (page 41). The -9 option was specified for **GZIP** and **BZIP2**, while **PPMD** was tested using 255 MiB of memory and a seventh order model.

The experiments in this section are first reported for character-based **RE-PAIR** for **WSJ508**, and then for punctuation-aligned **RE-PAIR** for **NEWS**. In both cases, the block size of **RE-PAIR** is 20,971,520 symbols, in order to provide some consistency with previous experiments.

As shown in Figure 5.2 on page 114, five combinations of the three phases of **RE-MERGE** are possible, labelled method A to E. As a consequence of the choices made during

<table>
<thead>
<tr>
<th></th>
<th><strong>WSJ508</strong></th>
<th></th>
<th><strong>NEWS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>compression ratio (bpc)</td>
<td>encoding time (s)</td>
<td>decoding time (s)</td>
</tr>
<tr>
<td><strong>GZIP</strong></td>
<td>2.941</td>
<td>170.0</td>
<td>20.4</td>
</tr>
<tr>
<td><strong>BZIP2</strong></td>
<td>2.105</td>
<td>447.3</td>
<td>155.8</td>
</tr>
<tr>
<td><strong>PPMD</strong></td>
<td>1.637</td>
<td>800.9</td>
<td>821.5</td>
</tr>
</tbody>
</table>

Table 5.5: Compression effectiveness and execution times for the compression of **WSJ508** and **NEWS** using **GZIP**, **BZIP2**, and **PPMD**. These results are taken directly from Table 2.5 (page 41) and Table 2.6 (page 41).
Block Merging for Re-Pair

the implementation of Re-Merge, the phrase hierarchy is encoded at the conclusion of every Re-Merge phase, and decoded at the beginning. However, the encoding of the modifiers and the reduced word sequence is applied after all necessary block merging phases have been completed. In these experiments, entropy coding is handled by Shuff, as was also the case in previous chapters. Each stream is processed by Shuff as a single block, except for the reduced sequence created by Re-Pair which contains \( b \) blocks. In this scenario, each of the \( b \) blocks are encoded by Shuff individually.

Some statistics about the phrase hierarchy and reduced sequence after Re-Pair and the five methods of Re-Merge are shown in Table 5.6. Most notable are changes between columns of the table. From Re-Pair to method A (Re-Merge phase 1 only), the size of the phrase hierarchy decreases dramatically as the blocks are merged. The maximum symbol generation in a block (gen) drops from 31 to 28 as duplicates are removed and the symbol generations collapse. Methods B and C, which end with Re-Merge phases 2a and 2b respectively, make no changes to the phrase hierarchy, but the sequence lengths are shortened. As expected, since phase 2b is tailored for effectiveness in reducing the length of the sequence, method C achieves a slightly shorter sequence than method B. Likewise, when these two methods are followed by phase 3 (method D for phase 2a and method E for phase 2b), method E is able to produce a shorter sequence with less phrases. With WSJ508 and character-based Re-Pair, Re-Merge offers no changes to the maximum symbol generation in a block, nor the length of the longest symbol \( (\alpha_L) \). Not shown in the table, the average distance between cut vertices for phase 2b was 66.4 primitives.

Compression ratios for character-based Re-Pair and the five combinations of Re-Merge are reported in Table 5.7. The baseline values for character-based Re-Pair are in line with those presented in Table 5.6. Between the column “None” and the column marked method A, a drop in the size of the phrase hierarchy occurs. Methods B and C improve compression, without changing the cost of the phrase hierarchy. The compression ratios of methods B and C improve again after phase 3, producing the results in columns D and E, respectively. In these last two columns, the overall compression effectiveness again improves, but at the cost of a larger phrase hierarchy.
5.6 Experiments

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>RE-MERGE Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>\Sigma</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>\rho</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>S'</td>
<td></td>
</tr>
<tr>
<td>gen</td>
<td>31</td>
<td>28</td>
</tr>
<tr>
<td>$</td>
<td>\alpha_L</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>26</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.6: Statistics for RE-MERGE with character-based RE-PAIR on the WSJ508 file. The column marked “None” represents no block merging performed, and $|\Sigma|$, $|\rho|$, and $|S'|$ represent the primitives, phrases, and final sequence, respectively. The row marked “gen” is the maximum symbol generation in any block, while $|\alpha_L|$ is the length of the longest phrase, expressed in primitives. The last row, $b$, is the number of blocks.

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>RE-MERGE Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phrase hierarchy</td>
<td>0.223</td>
<td>0.088</td>
</tr>
<tr>
<td>Sequence (SHUFF)</td>
<td>1.583</td>
<td>1.643</td>
</tr>
<tr>
<td>Total</td>
<td>1.806</td>
<td>1.731</td>
</tr>
</tbody>
</table>

Table 5.7: Compression ratios for character-based RE-PAIR and RE-MERGE for WSJ508, expressed in bits per character relative to the original size of the file.

The results for method E have been expanded into a graph as Figure 5.10, clearly separating the five iterations of phase 1 of RE-MERGE, and then the gain accrued from phase 2b and phase 3. Each iteration of phase 1 gradually decreases the size of the phrase hierarchy.

The compression, decompression, and block merging times for character-based RE-PAIR with WSJ508 are shown in Table 5.8. The time for phase 1 includes all 5 iterations necessary to transform the 26 blocks into a single block. Of interest is the fact that phase 2b takes 10 times longer than phase 2a. However, when phase 3 follows either (methods D and E), the time required is similar, regardless of which version of phase 2 has been chosen. In comparison to RE-PAIR and RE-MERGE, the time taken for sequence coding with SHUFF is small. As for decoding, note that regardless of which type of encoding has been chosen, the decoding requirements is small and always between 70 and 80 seconds. While phase 2b requires an average of 3,276.5 seconds, a simpler approach that eliminated secondary prefix symbols and extended symbols required an average of 5,994.5 seconds.
Block Merging for Re-Pair

![Graph](image.png)

Figure 5.10: Compression ratios of character-based Re-PAIR and method E of Re-MERGE for WSJ508. Each vertical bar represents the initial compression with Re-PAIR, or a single application of Re-MERGE. Initially, there are 26 blocks, which requires phase 1 five passes to combine into a single block. Then, phase 2b is applied, followed by phase 3.

This less principled approach retained an index to the primitives and primary prefix symbols, though.

The results from applying punctuation-aligned Re-PAIR to NEWS are similar to those of character-based Re-PAIR to WSJ508. The first step in punctuation-aligned Re-PAIR is the pre-processing stage with Pre-PAIR. When a Re-MERGE configuration has been chosen, the only difference between it and any other combination is the execution times and compression ratios of the word sequence. So, values related to the lexicons and the modifiers are constant. These fixed values are shown in Table 5.9 and re-appear in future tables in this chapter under the heading of “Other”.

Some phrase hierarchy and sequence statistics pertaining to punctuation-aligned Re-PAIR for NEWS are shown in Table 5.10. Re-PAIR creates nine blocks from the 168,721,097 symbols of the word sequence. As with character-based Re-PAIR, between the column
5.6 Experiments

<table>
<thead>
<tr>
<th>Encoding time</th>
<th>None</th>
<th>RE-MERGE Method</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE-PAIR</td>
<td>2,323.3</td>
<td>2,323.3</td>
<td>2,323.3</td>
<td>2,323.3</td>
<td>2,323.3</td>
<td>2,323.3</td>
<td></td>
</tr>
<tr>
<td>RE-MERGE 1</td>
<td>453.0</td>
<td>453.0</td>
<td>453.0</td>
<td>453.0</td>
<td>453.0</td>
<td>453.0</td>
<td></td>
</tr>
<tr>
<td>RE-MERGE 2a</td>
<td>394.4</td>
<td>394.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE-MERGE 2b</td>
<td>3,276.5</td>
<td>3,276.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE-MERGE 3</td>
<td></td>
<td></td>
<td>222.2</td>
<td>202.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHUFF</td>
<td>39.2</td>
<td>32.1</td>
<td>31.0</td>
<td>30.6</td>
<td>34.3</td>
<td>32.7</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2,362.5</td>
<td>2,808.4</td>
<td>3,201.7</td>
<td>6,083.4</td>
<td>3,427.2</td>
<td>6,288.0</td>
<td></td>
</tr>
</tbody>
</table>

Decoding time

| SHUFF | 9.3  | 8.1  | 8.3  | 8.9  | 9.3  | 9.2  |
| DES-PAR | 62.9 | 61.9 | 63.3 | 65.8 | 68.2 | 68.9 |
| Total  | 72.2 | 70.0 | 71.6 | 74.7 | 77.5 | 78.1 |

Table 5.8: Compression times for RE-MERGE with character-based RE-PAIR on WSJ508. In the first column, “None” represents no block merging performed, using the output from RE-PAIR directly. Times are in CPU seconds on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache, and averaged over three trials.

<table>
<thead>
<tr>
<th>Pre-PAR</th>
<th>Compression (bpc)</th>
<th>Encoding time (s)</th>
<th>Decoding time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streams:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word lexicon</td>
<td>0.005</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>Non-word lexicon</td>
<td>&lt;0.001</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>Case-folding modifiers</td>
<td>0.240</td>
<td>16.2</td>
<td>7.1</td>
</tr>
<tr>
<td>Stemming modifiers</td>
<td>0.377</td>
<td>18.3</td>
<td>7.5</td>
</tr>
<tr>
<td>Non-word modifiers</td>
<td>0.342</td>
<td>16.6</td>
<td>7.3</td>
</tr>
<tr>
<td>Total</td>
<td>0.964</td>
<td>777.2</td>
<td>165.1</td>
</tr>
</tbody>
</table>

Table 5.9: Compression ratios and execution times for the streams of Pre-PAR, not including the word sequence stream, when applied to NEWS. The lexicons have been front-coded and then compressed using bzip2. The two streams of modifiers and the non-word sequence have been compressed with shuff, using a block size of 20,971,520 symbols. Execution times have been averaged over three trials.

“None” and method A, the phrase hierarchy’s size decreases. Methods B and C both reduce the length of the sequence, with method C being more effective. Likewise, method E is an improvement to method D in both the final sequence length and the number of phrases required to achieve that length. With NEWS, there are no changes to the maximum
Block Merging for RE_PAIR

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>RE-MERGE Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>(</td>
<td>\Sigma</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>\rho</td>
<td>)</td>
</tr>
<tr>
<td>(</td>
<td>S'</td>
<td>)</td>
</tr>
<tr>
<td>gen</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>(</td>
<td>\alpha_L</td>
<td>)</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.10: Statistics for RE-MERGE with punctuation-aligned RE_PAIR. The column marked “None” represents no block merging performed. Primitives, phrases, and the final sequence are abbreviated as \(\Sigma\), \(\rho\), and \(S'\). The row marked “gen” is the maximum symbol generation in any block, while \(|\alpha_L|\) is the length of the longest symbol, expressed in primitives. The last row, \(b\), is the number of blocks.

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>RE-MERGE Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Phrase hierarchy</td>
<td>0.163</td>
<td>0.116</td>
</tr>
<tr>
<td>Sequence (SHUFF)</td>
<td>1.145</td>
<td>1.181</td>
</tr>
<tr>
<td>Other</td>
<td>0.964</td>
<td>0.964</td>
</tr>
<tr>
<td>Total</td>
<td>2.272</td>
<td>2.261</td>
</tr>
</tbody>
</table>

Table 5.11: Compression ratios for RE-MERGE with punctuation-aligned RE_PAIR. In the second column, “None” represents no block merging performed, using the output from RE_PAIR directly. In the sixth row, “Other” represents the compression ratio of the other streams, as shown in Table 5.9.

...generation in a block and the length of the longest symbol. The average length of a segment during phase 2b was 5.5 primitives, much less than for character-based RE_PAIR.

The compression ratios of Table 5.11 reflect the levels for character-based RE_PAIR, with one notable exception. While phase 3 has reduced the length of the sequence, the compression ratios of methods D and E are worse than those of methods B and C. To be precise, the compression ratios of the sequence for methods D and E have improved, but not enough to compensate for the increase in size of the phrase hierarchy.

The execution times for punctuation-aligned RE_PAIR and RE-MERGE are presented in Table 5.12. For the total times, the costs of encoding and decoding the modifiers have been taken into account in the rows marked “Other”. The execution time difference between methods are similar to the differences between methods for character-based RE_PAIR. And, as before, the decoding times are fairly consistent, whether RE_PAIR alone...
5.6 Experiments

<table>
<thead>
<tr>
<th>Encoding time</th>
<th>None</th>
<th>RE-MERGE Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Other</td>
<td>777.2</td>
<td>777.2</td>
</tr>
<tr>
<td>RE-PAIR</td>
<td>1,634.3</td>
<td>1,634.3</td>
</tr>
<tr>
<td>RE-MERGE 1</td>
<td>636.7</td>
<td>636.7</td>
</tr>
<tr>
<td>RE-MERGE 2a</td>
<td>1,142.2</td>
<td>1,142.2</td>
</tr>
<tr>
<td>RE-MERGE 2b</td>
<td>2,645.4</td>
<td>2,645.4</td>
</tr>
<tr>
<td>SHUFF</td>
<td>55.0</td>
<td>51.4</td>
</tr>
<tr>
<td>Total</td>
<td>2,466.5</td>
<td>3,099.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decoding time</th>
<th>SHUFF</th>
<th>DES-PAIR</th>
<th>Other</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHUFF</td>
<td>13.3</td>
<td>12.3</td>
<td>12.4</td>
<td>12.9</td>
</tr>
<tr>
<td>DES-PAIR</td>
<td>52.8</td>
<td>53.3</td>
<td>54.4</td>
<td>55.5</td>
</tr>
<tr>
<td>Other</td>
<td>165.1</td>
<td>165.1</td>
<td>165.1</td>
<td>165.1</td>
</tr>
<tr>
<td>Total</td>
<td>231.2</td>
<td>230.7</td>
<td>231.9</td>
<td>233.5</td>
</tr>
</tbody>
</table>

Table 5.12: Compression times for RE-MERGE with punctuation-aligned RE-PAIR for the file NEWS. In the first column, “None” represents no block merging performed, using the output from RE-PAIR directly. The rows labelled “Other” represent the total compression times of the other streams, as shown in Table 5.9. Times are in CPU seconds on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache.

has been applied, or followed by a RE-MERGE method.

The elementary approach to phase 2b was also applied, and was actually found to be marginally faster. Without secondary prefix symbols and extended symbols, 2,569.7 seconds was taken by RE-MERGE, as opposed to 2,640.9 seconds. The time to construct these symbol links for phase 2b was 38.7 seconds, on average. There was no gain in time by adding these links partly due to the generally shorter symbols than character-based RE-PAIR. The longest symbol for punctuation-aligned RE-PAIR had 1,348 word tokens, while the longest symbol for character-based RE-PAIR consisted of 4,970 characters. In comparison, the extra symbol linking time for character-based RE-PAIR was 9.2 seconds, which was easily justifiable due to the improvements in speed. While the phrase hierarchy produced by punctuation-aligned RE-PAIR was larger, since the primitives represent words, there was less similarity between consecutive symbols in the phrase hierarchy sorted by expanded strings. So, if memory constraints exists, then choosing the simpler version
of phase 2b would save $6(|\Sigma| + |\rho|)$ words of memory for punctuation-aligned RE-PAIR.

5.7 Append to a Compressed Document

The success of RE-MERGE as a post-processor for RE-PAIR is due to its memory usage. While RE-PAIR requires enough space to hold the entire message, each phase of RE-MERGE occupies memory which is proportional to the size of the phrase hierarchy. In the experiments described above, WSJ508 and NEWS could be processed by RE-MERGE on a test machine with 1 GiB of memory.

Problems with RE-MERGE arise if the phrase hierarchy of the reduced message is larger than the amount of available memory. This section proposes a solution which involves the creation of a phrase hierarchy from a subset of the message to act as training data. The size of the subset must be small enough so that when it is post-processed with RE-MERGE, the phrase hierarchy can be kept within memory. This phrase hierarchy is then used to compress the remainder of the message through a technique similar to phase 2b. Hence, the time and space requirements for appending text is same as for phase 2b.

Appending text to a compressed document presents two problems. First, the phrases created with the training data may not be representative of the rest of the message. As more text is processed with the static phrase hierarchy, compression is expected to degrade. Second, RE-MERGE requires the set of primitives to be known ahead of time. Regardless of how much of a message has been seen, the appearance of a novel primitive can always occur. The structure of the phrase hierarchy and the method with which it is encoded prevents symbols to be inserted in the middle. Whenever a new symbol is added to the phrase hierarchy, the structure must be re-built and the symbols in the sequence renumbered, as with phase 3 of RE-MERGE.

The experiments described below respond to the first problem by demonstrating how the size of the training data affects overall compression effectiveness. Appending text with RE-MERGE was implemented specifically for character-based RE-PAIR. As a result, the set of primitives is limited to the 256 characters of the Latin-1 character set. After
the experiments, a proposed solution for word-based and punctuation-aligned RE-PAIR is discussed.

Appending text with RE-MERGE is similar to the semi-static version of RAY called XRAY [Cannane and Williams, 2002]. The XRAY algorithm creates a dictionary using a portion of the message as training data. Then, this training data is applied on-line to the remainder of the message. After the training phase, Huffman codes are assigned to the dictionary entries, so that phrase selection afterwards is based on the lengths of the corresponding codewords. A window of text is examined at a time, and phrases are chosen based on the assignment of a penalty for bypassing a replacement. In contrast to XRAY, RE-MERGE continues to separate the phrase selection heuristic from the entropy coder for the reduced word sequence, so that phrases are selected based on how much the sequence length can be reduced. As with phase 2b, each primitive from the second part of the message is placed in a node in a DAG, and the shortest path through is determined by following edges representing phrase hierarchy symbols.

In order to measure the performance of appending text, character-based RE-PAIR was used to compress WSJ508. Various sizes of training data were taken from the front of WSJ508 and partitioned into blocks of 20 MiB each. These blocks were merged using method E of RE-MERGE. That is, all of the blocks were combined through multiple iterations of phase 1, followed by phase 2b, and then phase 3. The remainder of WSJ508 is then appended to the end of the reduced training data. An alternative experimental framework would obtain the training data from random sections within WSJ508. While the current implementation of RE-MERGE would support this approach, it was not used because the articles within WSJ508 are ordered chronologically. Appending text gives the illusion that new documents are being added to the collection over time.

Table 5.13 provides statistics from these experiments after all of WSJ508 has been processed. For each training size ranging from 20 MiB to 400 MiB, the number of primitives, the number of phrases, the length of the final sequence, the length of the longest phrase in primitives, and the maximum generation are presented. The last row of the table is obtained from earlier experiments with character-based RE-PAIR and WSJ508 (see the col-
Block Merging for Re-Pair

| Training size (MiB) | $|\Sigma|$ | $|\rho|$ | $|\mathcal{S}|$ | $|\alpha_L|$ | gen |
|---------------------|-----------|-----------|---------------|--------------|-----|
| 20                  | 256       | 322,432   | 71,240,604    | 1,257        | 21  |
| 40                  | 256       | 552,784   | 65,724,383    | 1,257        | 22  |
| 80                  | 256       | 948,903   | 60,802,497    | 2,084        | 21  |
| 120                 | 256       | 1,302,459 | 58,221,318    | 4,970        | 21  |
| 160                 | 256       | 1,647,723 | 55,121,237    | 4,970        | 23  |
| 240                 | 256       | 2,278,342 | 52,039,645    | 4,970        | 24  |
| 320                 | 256       | 2,906,923 | 44,730,578    | 4,970        | 27  |
| 400                 | 256       | 3,467,728 | 40,738,832    | 4,970        | 28  |
| 508                 | 94        | 4,164,166 | 38,955,830    | 4,970        | 28  |

Table 5.13: Statistics for appending text with character-based Re-PAIR on WSJ508. The first column indicates the size of the training data. Then, $\Sigma$, $\rho$, and $\mathcal{S}$ represent the primitives, phrases, and final sequence, respectively. The column marked $|\alpha_L|$ is the length of the longest phrase in primitives, and “gen” is the maximum symbol generation.

...umn marked “method E” in Table 5.6). Those experiments are equivalent to making all of WSJ508 the training data.

As Table 5.13 shows, as more training data is used, the number of phrases, the maximum phrase length, and the maximum generation increase. Also, since the phrase hierarchy contains more symbols, the length of the sequence becomes shorter. Note that the number of primitives is always fixed at 256. In the last row, when all of the message is seen, the space allocated to generation 0 is equal to the number of distinct primitives that appear in the message.

Figure 5.11 presents the compression levels achieved as a function of the size of the training data. The compressed sizes are calculated from the merged training data, and again once the rest of WSJ508 has been appended. In both cases, the reduced sequence was encoded with SHUFF as a single block. As expected, overall compression effectiveness improves as the training set increases in size. The best compression ratio on the graph is achieved when the initial training size is equal to the size of WSJ508. However, with just 16% of WSJ508 used as training data, the compression effectiveness achieved is only 18% worse than applying Re-PAIR and Re-MERGE on the entire file.

The time required to compress WSJ508 is presented in Table 5.14. As the size of the training data increases, the time required to apply both Re-PAIR and method E of...
5.7 Appending to a Compressed Document

![Graph showing compression effectiveness over training data size](image)

Figure 5.11: Compression effectiveness of Re-MERGE when appending text to various sizes of training data using character-based Re-PAIR and the WSJ508 test file. The initial block size for Re-PAIR is 20 MiB. The solid line represents the compression effectiveness of the entire file, while the dashed line indicates the size of the compressed training data.

Re-MERGE also increases. While Re-MERGE is not necessary for the training size of 20 MiB, it was applied in order to remove any duplicates in the phrase hierarchy. The time required to append the remaining text does not increase in a similar manner, though. After a certain training size, the cost of appending is offset by the amount of text left to add. The time required to append text peaks when the training size is 80 MiB. The time taken to encode the reduced sequence with SHUFF is similar throughout the experiment. Regardless of the costs of appending and sequence encoding, the dominant times belong to Re-PAIR and method E of Re-MERGE. Because of this, the total time increases with the training size.

The decompression times for appending text are given in Table 5.15. The decoding time of SHUFF and the phrase expansion time of Des-PAIR both increase as the training data becomes larger. This increase is due to the larger phrase hierarchy, despite the sequence becoming shorter.

The addition of text to a compressed message serves two purposes. The first, as mentioned already, is to enable documents which yield a large phrase hierarchy to be processed. The second purpose is to support dynamic collections, where new documents are added to the repository throughout its lifetime. The coding mechanism provided
Block Merging for Re-Pair

<table>
<thead>
<tr>
<th>Training size (MiB)</th>
<th>RE-PAIR</th>
<th>RE-MERGE (method E)</th>
<th>Append text</th>
<th>SHUFF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>88.7</td>
<td>114.6</td>
<td>2,662.3</td>
<td>31.6</td>
<td>2,897.2</td>
</tr>
<tr>
<td>40</td>
<td>180.0</td>
<td>186.3</td>
<td>2,937.7</td>
<td>31.3</td>
<td>3,335.3</td>
</tr>
<tr>
<td>80</td>
<td>363.0</td>
<td>428.1</td>
<td>3,165.7</td>
<td>31.5</td>
<td>3,988.3</td>
</tr>
<tr>
<td>120</td>
<td>546.4</td>
<td>708.5</td>
<td>3,126.0</td>
<td>31.6</td>
<td>4,412.5</td>
</tr>
<tr>
<td>160</td>
<td>732.3</td>
<td>981.4</td>
<td>2,986.1</td>
<td>31.8</td>
<td>4,731.6</td>
</tr>
<tr>
<td>240</td>
<td>1,102.5</td>
<td>1,610.9</td>
<td>2,457.6</td>
<td>32.0</td>
<td>5,203.0</td>
</tr>
<tr>
<td>320</td>
<td>1,467.2</td>
<td>2,246.2</td>
<td>1,891.2</td>
<td>31.6</td>
<td>5,636.2</td>
</tr>
<tr>
<td>400</td>
<td>1,834.2</td>
<td>2,954.3</td>
<td>1,200.5</td>
<td>31.6</td>
<td>6,020.6</td>
</tr>
<tr>
<td>508</td>
<td>2,323.3</td>
<td>3,932.0</td>
<td>—</td>
<td>32.7</td>
<td>6,288.0</td>
</tr>
</tbody>
</table>

Table 5.14: Encoding times for character-based Re-PAIR and appending with Re-MERGE for various sizes of training data. Times are in CPU seconds and averaged over three trials on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache.

<table>
<thead>
<tr>
<th>Training size (MiB)</th>
<th>SHUFF</th>
<th>DES-PAIR</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>7.0</td>
<td>56.0</td>
<td>63.0</td>
</tr>
<tr>
<td>40</td>
<td>7.3</td>
<td>57.4</td>
<td>64.7</td>
</tr>
<tr>
<td>80</td>
<td>7.7</td>
<td>59.7</td>
<td>67.4</td>
</tr>
<tr>
<td>120</td>
<td>7.8</td>
<td>61.3</td>
<td>69.1</td>
</tr>
<tr>
<td>160</td>
<td>8.1</td>
<td>62.2</td>
<td>70.3</td>
</tr>
<tr>
<td>240</td>
<td>8.3</td>
<td>62.9</td>
<td>71.2</td>
</tr>
<tr>
<td>320</td>
<td>8.6</td>
<td>64.4</td>
<td>73.0</td>
</tr>
<tr>
<td>400</td>
<td>8.7</td>
<td>65.8</td>
<td>74.5</td>
</tr>
<tr>
<td>508</td>
<td>9.2</td>
<td>68.2</td>
<td>78.1</td>
</tr>
</tbody>
</table>

Table 5.15: Times for decoding a compressed message made with character-based Re-PAIR with various sizes of training data. Times are in CPU seconds and averaged over three trials on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache.

by SHUFF does not allow periodic additions to the compressed sequence, and the solution adopted by the experiments is to re-encode the sequence after any changes. An alternative is to apply SHUFF in blocks, and make each extension a block. Otherwise, support of both aims for character-based Re-PAIR is complete.

The problem becomes more complicated for word-based and punctuation-aligned Re-PAIR, since no restriction is placed on the number of unique words. When a novel word appears, it must be added to the word lexicon. However, since the positions in the lexically sorted word lexicon are used to create the word sequence for Re-PAIR, both streams must
be recompressed. A similar problem occurs when a novel non-word symbol appears in the message.

Moffat et al. [1997] discusses several solutions for compressing dynamic document databases, two of which can be adapted for RE-PAIR. The first idea adds an escape symbol which signals to the decoder that the following is a novel word, specified using a secondary model, such as characters. While this idea permits dynamic document compression, the loss of the word boundaries enforced by PRE-PAIR make it unsuitable for browsing. The second method proposed by Moffat et al. would also introduce an escape primitive to the phrase hierarchy, except that the escape symbol forces the decoder to consult an auxiliary word lexicon. This secondary word lexicon is unsorted, and no limit is placed on its size. However, because of the compact nature of RE-PAIR’s phrase hierarchy, these words cannot be used to form phrases. One way of circumventing this problem is to extend the escape mechanism to phrases. Whenever an escape is encountered while expanding the reduced word sequence, the auxiliary phrase hierarchy is consulted instead.

The ideal method of compression for the unstructured auxiliary phrase hierarchy is undetermined at this time and appending of text with word-based or punctuation-aligned RE-PAIR requires further investigation.

## 5.8 Summary

This chapter has described RE-MERGE as a post-processing step to RE-PAIR. When the source message is too large, it must be compressed as independent blocks by RE-PAIR and then merged with RE-MERGE. RE-MERGE is feasible because each phase requires memory proportional to the size of the phrase hierarchy. This is in contrast to RE-PAIR, whose memory usage is related to the length of the message. Even so, a message could be so large that its corresponding phrase hierarchy exceeds the amount of memory available. This problem was considered for character-based RE-PAIR. Adapting this solution for word-based and punctuation-aligned RE-PAIR is reserved for future work.

The block merging techniques of this chapter serve three major goals. The first goal is to enable phrase browsing by combining multiple phrase hierarchies into a single one. The
Block Merging for Re-Pair

second goal is to improve compression effectiveness through the removal of duplicates in the phrase hierarchy and the identification of old and new symbol pairs for replacement. In satisfying the second goal, a third objective emerges, since the quality of the symbols in the phrase hierarchy is improved as more symbol pairs are replaced.

The experiments in this chapter have shown that to achieve these goals, no single best combination of Re-MERGE phases exists. The three phases of Re-MERGE have been organised into five methods which illustrate the trade-offs between program execution time and compression effectiveness. All five methods satisfy the first goal by outputting a unified phrase hierarchy. However, achieving the second and third targets requires considerable additional computation time. From the perspective of phrase browsing, a reduced message that has been post-processed by any of the five methods of Re-MERGE can be browsed. Which method should be chosen depends on the amount of time available, and on the compression effectiveness and phrase hierarchy quality that is required.

The results from the experiments with punctuation-aligned Re-PAIR and NEWS have been summarised in Figure 5.12. The graph plots the compression effectiveness of Re-PAIR and the five methods of Re-MERGE against their throughput (number of bytes processed per second).

The emphasis in this thesis is on phrase browsing, and for the remainder of this investigation method E for punctuation-aligned Re-PAIR is the preferred approach. While this method is expensive with respect to compression speed, decompression time is unaffected by this decision.

Chapters 3, 4, and 5 have not considered sequence and modifier coding. The coding of these two types of streams is addressed in Chapter 6, by first considering the problems inherent in the method chosen so far (Huffman coding with SHUFF) and then by proposing two alternative coding schemes.
Figure 5.12: Graph of compression effectiveness versus throughput for the previous section’s experiments with punctuation-aligned Re-Pair on the news file for the five methods of Re-MERGE. Times are for encoding, and are measured on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache.
Chapter 6

Coding Considerations

Chapter 2 explained how compression systems can be thought of as three separate components: modelling, probability estimation, and coding (see Figure 2.1, page 17). In the chapter that followed, the RE-PAIR algorithm was demonstrated as a method of reducing the length of a message through the recursive replacement of character digrams. Later, pre-processing and post-processing stages were introduced which, when working with RE-PAIR, provided a combination of tools for parsing a document into words, reducing the length of the sequence of word tokens, and finally merging blocks so that larger messages can be handled. The compression aspects from Chapter 3 to Chapter 5 have placed emphasis on the modelling and probability estimation components of a compression system.

A relatively small amount of attention was given to the third component: coding. The goal of this chapter is to rectify that omission by considering several methods of coding, and concluding with a selection of coders which follows the theme of this thesis. At the end of this chapter, a coding scheme which provides a suitable balance between compression and browsing for punctuation-aligned RE-PAIR is described.

The chapter is structured as follows. Section 6.1 summarises the thesis so far with respect to coding. Section 6.2 and 6.3 propose two coding mechanisms. The first builds on the Huffman coder that has already been used throughout this thesis, while the second relaxes the requirements of entropy coding altogether in order to achieve more efficient phrase browsing. The effectiveness and efficiency of these two coding mechanisms are described in Section 6.4. Section 6.5 concludes this chapter with an overview of the
advantages and disadvantages of the coding methods that have been considered.

6.1 Coding for Compression and Phrase Browsing

The coding techniques employed in previous chapters have been biased towards compression in that they favour compression effectiveness over other more pragmatic factors. This chapter introduces alternatives which attempt to compromise between compression and the special needs of phrase browsing. Before examining these alternatives, a brief review of the previous chapters with respect to coding is appropriate.

The coding component of a compression system encodes the output of the probability estimator into a bit stream for transmission. Improved compression effectiveness is achieved when the number of bits required is reduced. The least number of bits possible for losslessly encoding the message is defined by the self-information (or self-entropy) of the message, as covered in Chapter 2. An approach which attempts to achieve a compression ratio close to the self-information is an entropy coder.

While there are several factors to consider when deciding on a coder for a compression system, the use of an entropy coder implies that some importance has been given to compression effectiveness. The need to permit phrase browsing in the compressed message shifts the importance to decoding time and searchability. In any interactive system, a timely response to the user is essential. Sacrificing a controlled amount of compression effectiveness might be justified, if a shorter response time can be assured.

In the case of punctuation-aligned (or word-based) RE-PAIR, seven streams are produced by PRE-PAIR, RE-PAIR, and RE-MERGE. These streams serve different purposes, but they can be divided into two broad categories based on their coding requirements for phrase browsing. There are the data streams that need to be decoded completely in order to be useful, and there are those that need to be only partially decoded if only a fraction of the original message is to be presented.
6.1 Coding for Compression and Phrase Browsing

The Phrase Hierarchy and Lexicons

The first category of streams includes the word lexicon, the phrase hierarchy, and the non-word lexicon. The two most important streams for phrase browsing are the word lexicon and the phrase hierarchy. The word lexicon lists every unique word token found in the document, while the phrase hierarchy contains information about the relationships formed between the word tokens as they form longer phrases. Separate from these two streams is the non-word lexicon, which is not required for phrase browsing. However, the commonality that exists between all three of these streams is that they are encoded so that compression effectiveness is favoured. As a result, the compression mechanisms applied require that they must be completely decoded in order to be useful.

The two lexicons are encoded with front-coding, and can then be processed by a standard compression tool such as BZIP2. The two-dimensional phrase hierarchy is mapped to a single dimension via the chiastic slide before being encoded with interpolative coding. These two methods of encoding are biased towards compression effectiveness. But the strings involved have to be fully decoded to support phrase browsing, so this approach is acceptable. When these compression methods are applied, these three streams were small in comparison to the remaining four. For example, the 1,000 Mbi of NEWS was processed with punctuation-aligned RE-PAIR and method E of RE-MERGE in Section 5.6, these three streams represent 6.4% of the total compressed message, or 0.144 bits per character with respect to the original message (see Table 5.9 on page 147 and Table 5.11 on page 148).

The Modifiers and Reduced Sequence

The other four streams possess a much simpler structure than the phrase hierarchy or the lexicons. Each stream is composed of symbols which can be arbitrarily decoded without examining neighbouring symbols. A symbol is represented as a number, which refers to an entry in the non-word lexicon, a set of instructions to reverse stemming or case folding, or a symbol in the phrase hierarchy.
Coding Considerations

Strictly speaking, in each stream, a number (or reference) does not appear independently of its neighbours. References in the reduced word sequence do not appear at random with equal probability because after Re-Pair (and Re-Merge), no pair of adjacent symbols appear twice or more in the entire sequence. One could also argue that the appearance of certain combinations of case folding modifiers precludes the same or some other type immediately after. For example, if the first letter of a word token is in upper case, it is probably signalling the beginning of a new sentence, and one would expect the previous non-word to include a period, and the next few words to be entirely in lower case. When the modifier streams were first discussed in Chapter 4, results from experiments with higher-order models were shown (Table 4.10 on page 97). However, in the interest of obtaining a modifier from any position in the stream without decoding everything before it, a zero-order model was advocated in Chapter 4. Because of this, coding was handled by the Huffman coder, SHUFF.

Two entropy coding schemes for a sequence of references were discussed in Chapter 2. They were arithmetic coding and Huffman coding, with available implementations being UINT and SHUFF, respectively. Chapter 2 and Chapter 3 demonstrated that UINT achieved better overall compression effectiveness, but required more time for both encoding and decoding. Moffat and Zobel [1992] argued that arithmetic coding is unsuitable for an information retrieval system due to the decoding time involved, which translates to waiting time for the user. Instead, Huffman coding was employed and was shown to yield a small loss in compression effectiveness, while providing an estimated improvement of 40 times over an implementation of arithmetic coding by Witten et al. [1991]. Following the work by Moffat and Zobel, UINT was abandoned as a potential coding candidate in a phrase browsing system with Re-Pair. The systems detailed since Chapter 3 have only considered SHUFF as the coder for Re-Pair’s modifier streams and reduced sequence.

Unfortunately, applying the Huffman coder SHUFF to these four streams does not meet the needs of phrase browsing. To illustrate this, a description of how the streams need to be accessed by a phrase browser is necessary. The phrase hierarchy and the word lexicon allow the user to manoeuvre around the primitives and phrases, with the exact
steps performed being the topic of Chapter 7. Phrase browsing ceases when the user has decided on a symbol of interest, \( \alpha \). The user will then examine the contexts in which \( \alpha \) occurs. Within the reduced word sequence, \( \alpha \) may appear directly or indirectly, as part of a symbol in a higher generation. That is, the browsing system must translate the symbol \( \alpha \) into a set of symbols \( C = (\alpha_1, \alpha_2, ..., \alpha_k) \) where \( \alpha = \alpha_1 \), but \( \alpha_2 \) to \( \alpha_k \) are higher-generation symbols which contain \( \alpha \). Precise details about the search through the phrase hierarchy is provided in the next chapter, but can be summarised as a recursive search for all symbols which contain \( \alpha \). The important point for this chapter is that a search for every symbol in \( C \) is done next on the reduced word sequence.

The locations of the symbols of \( C \) are determined and recorded in the set \( \mathcal{R} \), along with some suitable context for each. One plausible context may be the 10 symbols before and after each location. Results in \( \mathcal{R} \) are interactively shown to the user by using the phrase hierarchy and the word lexicon to translate the reduced word sequence references to words.

If the displayed result does not satisfy the user’s information need, further clarification may be required. The initial crude result displays stemmed, case folded word tokens with non-word tokens removed. To improve visual appeal, the initial display might add a space character between each word token. Ultimately, further refinement of the displayed result through the decoding of the three modifier streams is required, and if the \( i \)th symbol in \( \mathcal{R} \) (\( \mathcal{R}_i \)) is being displayed to the user, the corresponding modifiers for the symbol and its context need to be located and applied.

There is an important distinction between the purposes of the reduced word sequence and the three modifier streams. A set of symbols are searched for in the reduced word sequence, based on specified symbol values. The modifier streams are then synchronised to the same locations. The symbols in the modifier streams are located by their positions, and not their values. Also, locating modifiers for \( |\mathcal{R}| \) contexts is unnecessary, since not every result in \( \mathcal{R} \) may require clarification. So, one approach to the problem of modifier look-up is to obtain one position and its context at a time, as required.

With these points in mind, the next two sections look at ways of coding the four
Coding Considerations

streams to enable the necessary searching and synchronisation operations.

6.2 Huffman Coding in Blocks

Motivated by the discussion of compression effectiveness in Chapter 4, the application of several compression systems on the three modifier streams was considered and the costs in time and benefits in space were assessed. While compression effectiveness was good, there are two drawbacks to the approaches with respect to retrieval. First, because the user is waiting for the results, decoding should be fast. While previous results with dictionary-based systems such as GZIP and RE-PAIR have shown that decoding can be done efficiently, the time required cannot be ignored. Second, each time the modifier stream is accessed, only one position is sought. Once that symbol has been found, only it and its neighbouring symbols are required. Decoding the entire stream from the beginning until the symbol of interest seems wasteful.

If time is crucial, then a flat binary code can be employed, as shown in Table 2.4 on page 31. That is, each symbol in each of the modifier streams is encoded as an $L$-bit integer. The parameter $L$ is chosen so that no symbol in the stream is larger than $2^L$. Two passes over the stream are required for encoding, with the first pass used to determine $L$, and the second pass for encoding each symbol. Since every symbol is fixed at $L$ bits in width, any symbol can be retrieved provided $L$ is specified to the decoder. Retrieval is fast and is only limited by the physical characteristics of the storage medium.

In this section, a third alternative is presented which compromises between these extremes. Compression is achieved by encoding each stream directly with a zero-order Huffman coder (SHUFF). Unlike in earlier chapters, each stream is partitioned into blocks before being processed. In Chapter 5, a message was divided into blocks by RE-PAIR because of limitations in memory. Now, separating the streams of phrase numbers into blocks of $B$ symbols serves a second purpose – to allow any arbitrary block to be decoded by creating a partial index $I$, as shown in Figure 6.1. Each entry in the index points to the beginning of the corresponding block of Huffman coded symbol numbers.
6.2 Huffman Coding in Blocks

![Figure 6.1: The Huffman coding approach with an index which leads to the beginning of each block. Each block is self-contained, and includes the same number of coded symbols.](image)

**Algorithm 6.1:** Algorithm for Indexed SHUFF.

```plaintext
input : \( S \), block size \( B \)
output : \( S' \), index \( I \)
1 \( i \leftarrow 0; \ S' \leftarrow \{ \}; \ I \leftarrow \{ \} \)
2 for \( i < |S| \) do
3 \( \text{current}_\text{block} \leftarrow \text{encodeShuff}(S[i...i + B - 1]) \)
4 append \( \text{current}_\text{block} \) to \( I \)
5 append \( \text{current}_\text{block} \) to \( S' \)
6 \( i \leftarrow i + B \)
7 return \( (S', I) \)
```

The algorithm for this approach, called indexed SHUFF, is shown in Algorithm 6.1. Each block of \( B \) symbols is encoded with SHUFF in isolation and appended to the output stream, \( S' \). The size of each compressed block is appended to the index \( I \).

The number of blocks created for a stream of \( n \) symbols is \( \lceil n/B \rceil \). When a symbol is being sought, the fixed block size ensures that a position \( p \) in the stream must be in block \( [p/B] \). If line 4 of the algorithm is changed so that \( I \) contains cumulative compressed block sizes, then only two index entries need to be obtained in order to find the beginning and ending of the corresponding block. Otherwise, every index entry up until the two being sought need to be decoded. Within the compressed stream \( (S') \), position \( p \) plus some context to the left and right is retrieved. If the size of the window of symbols is smaller than \( B \), at most two adjacent blocks need to be accessed from disk, in addition to the entries in the index.

It is important to note that the SHUFF coder has the ability to divide an input stream into blocks, according to options provided by the user. However, the starting locations
Coding Considerations

of each block are invisible to the decoding program, preventing any block from being
processed without decoding all previous blocks.

Since each block of \( B \) symbols are compressed individually, indexed SHUFF allows the
input stream to be read in blocks of \( B \) symbols at a time. Indexed SHUFF is expected to
take longer than compressing the entire stream with SHUFF as a single block because of
the number of disk accesses necessary. Experiments later in this chapter survey the effect
various choices for \( B \) have on time and space.

6.3 Byte-Aligned Coding in Blocks

The reduced word sequence needs to support a search operation and not just the seek
mechanisms described in the previous section. For efficiency reasons, it is preferred that
the sequence be searched while in compressed form. Decoding the sequence before search-
ning requires extra time. Moreover, it may be more efficient to search through a compressed
message because it is smaller than its corresponding uncompressed form.

Motivated by these observations, Amir and Benson [1992] described the problem of
searching a message in compressed form and labelled it as the compressed matching prob-
lem. Since then, searching in a compressed message has received a significant amount of
attention, with the searching mechanism generally being directly linked to the compression
algorithm employed. Some recent work in this area include searching in data encoded with
Huffman coding [Klein and Shapira, 2001, Takeda et al., 2002] and searching in messages
which have been compressed with LZ78 [Navarro et al., 2001] and SEQUITUR [Mitarai
et al., 2001].

The problem considered by Klein and Shapira is that of searching in a Huffman coded
text. As Huffman codewords are variable length, a match may be a false one if it is not
aligned on a codeword boundary. The solution proposed by Klein and Shapira extends
earlier work which showed how Huffman codes tend to resynchronise after errors [Klein
and Wiseman, 2000]. If a possible match is found at position \( i \), then their algorithm jumps
back a constant number of bits and decodes all of the bits up until position \( i - 1 \) with
a Huffman tree. Probability analysis combined with experiments demonstrate that the
quality of the search results improves as the size of the section of bits decoded increases. However, their method is unsuitable for the reduced sequence because several patterns may need to be searched simultaneously. The number of symbols in $C$ can be large, depending on the symbol selected by the user, and the number of other symbols that rely on it. Ideally, a search method should make a single pass through the reduced sequence, regardless of the size of $C$.

Takeda et al. [2002] described a method of searching compressed documents without prior modification. They also demonstrated their technique on multi-byte character texts as well as semi-structured texts, and showed that any prefix-free code, including Huffman codes, could be searched. The basis of their idea was to combine synchronisation with string searching by constructing a pattern matching machine, which is then run on the entire file.

Manber [1997] and de Moura et al. [2000] looked at byte-aligned codewords to improve efficiency and to avoid the problem of aligning with codeword boundaries. Decoding byte-aligned codewords is fast, since repeated bit operations like bit shifting and bit masking are eliminated. Manber devised a mechanism for compressing and searching ASCII text which resembles the pairing of RE-PAIR. Since ASCII is limited to only 128 symbols, frequent pairs of symbols were replaced non-recursively until the compressed message had 256 unique symbols. By converting the search pattern in a similar fashion, the pattern is searched for directly in the compressed message using a string-matching technique such as Boyer and Moore [1977]. Alternatively, de Moura et al. implemented Huffman coding with a radix-256 code. That is, the nodes in the generated Huffman code tree had an out-degree of 256, instead of 2 for a binary code. Building upon this idea, a Tagged Huffman code was used for compression and searching. Tagged Huffman coding represented words as groups of 7 bit codewords (radix-128), with an eighth bit reserved for indicating the first byte of a codeword, to ensure alignment while searching. In their experiments, Tagged Huffman coding offered compression effectiveness which was around 5 bits per symbol worse than binary Huffman coding. However, overall search times required about half of the time of a direct search on the uncompressed message. More recently, Brisaboa
et al. [2003] extended this work by generalising Tagged Huffman codes to \((S, C)\)-Dense Codes. \((S, C)\)-Dense Codes consist of continuers and stoppers. Stoppers are bytes which indicate the last byte of a codeword. Other bytes are continuers. Tagged Huffman coding is a special case when there are 128 stoppers and 128 continuers. Brisaboa et al. varied the size of these two sets of codes to determine the combination which yielded the best compression ratio for various document collections.

The entropy coding requirements for the reduced word sequence are further eased compared to this previous work, in favour of retrieval. A coder called Re-View has been developed which operates with a corresponding codeword length limit \(L\). Blocks are created so that each block has at most \(2^L\) distinct values. Byte-alignment is achieved by restricting \(L\) to be either 8 or 16. These two variants are denoted as Re-View\(_8\) and Re-View\(_{16}\), respectively.

A sketch of a Re-View block and part of the following block is shown in Figure 6.2. Each block consists of a prelude and codewords, similar to a SHuff block. The prelude specifies the \(2^L\) different symbol values that exist in the block. At the beginning of each block, four values are also required, shown shaded in the figure.

The four values in the header provide information about the entire block to improve the search time of Re-View. In addition to the byte-alignment of codewords, the infor-
6.3 Byte-Aligned Coding in Blocks

mational headers allow a block to be bypassed when none of the symbols being sought (C) are known to exist inside. After phrase browsing, a user may have honed in on a phrase which is highly descriptive, and exists in only certain parts of the document collection. Areas in the collection which do not contain the phrase should be skipped. This function is provided by the last three of the shaded values of Figure 6.2. In order to determine whether or not a symbol in C exists in a block, the prelude must be decoded. If no symbols in C are found, then the block of codewords may be skipped, as shown by the solid, curved arrow in the figure. There are $B_k$ symbols in block $k$, which translates to $B_k(L/8)$ bytes. The number of distinct values in the block ($D_k$) is required to aid in decoding the prelude, and is $2^L$ for every block except for the last one. The number of original word tokens covered by a block is specified in the header as $W_k$, and is necessary for synchronisation with the modifier streams. After RE-PAIR, the number of word tokens represented by a block is equal to or greater than the number of symbols it contains. When a block is skipped, not only does the number of phrase symbols that have been passed need to be known ($B_k$), but the number of original word tokens as well. Then, within a block, as symbols are read, a count of the number of word tokens skipped and read since the beginning of the stream is maintained, so that each word token's corresponding modifier can be determined accurately. In order to support this requirement, both encoding and decoding of a reduced word sequence requires the lengths (in number of word tokens) of each symbol in the phrase hierarchy.

In later experiments, RE-VIEW is applied to the modifier streams as well, to assist in its understanding. As with indexed shuff, every block, except for the one which contains the position being sought, needs to be skipped. The first value in the header indicates the size of the prelude in bytes to allow both the prelude and the codewords of a block to be skipped, as shown by the dashed line of Figure 6.2. The block of interest can be located by summing $B_k$ for every block. In order to bypass the prelude and codewords of the current block $k$, the number of bytes is calculated as $P_k + B_k(L/8)$.

The algorithm for RE-VIEW is presented in Algorithm 6.2. RE-VIEW processes a stream ($S$) by scanning it sequentially while keeping track of the number of distinct
symbol values seen so far \((D_k)\). The distinct symbol values are maintained in an array 
\texttt{symbol_map}, whose size is proportional to the number of distinct symbols in the entire 
sequence (the size of the alphabet, \(|\Sigma|\)). Once \(2^L\) different values have been seen (line 9), 
a block is created. The distinct symbols that appear in the block are mapped to \(L\)-bit 
integers on line 14, and recorded in the prelude. The prelude is buffered so that \(P_k\) can be 
placed first within a block. The codewords are produced by mapping the symbol values 
to \(L\)-bit integers, by consulting the \texttt{symbol_map} array on line 22. The symbols in the 
block are represented as fixed \(L\)-bit codewords, and retain \(L\)-bit alignment throughout 
the block.

While \texttt{symbol_map} is scanned, each symbol is recorded in the prelude on line 13. 
Symbols are encoding as differences from the preceding symbol using nibble-aligned codes 
(4-bit codes). The mapping of differences to nibble-aligned codes is specified in Table 6.1. 
In the second column of the table, the codes employed are specified using hexadecimal.

The main data structures employed by \texttt{RE-View} are the sequence and the \texttt{symbol_map} 
array, requiring a total of \(|S| + |\Sigma|\) words of memory. Three main operations are performed 
by \texttt{RE-View} on these two structures. When a block is being built, two passes are 
performed over the reduced sequence. The first pass inspects the sequence in order to identify 
\(2^L\) distinct values. The second pass maps each symbol to an \(L\)-bit codeword. Also, in 
order to assign symbols to codewords, a single pass over the \texttt{symbol_map} structure is neces-
sary for each of the \(K\) blocks created. Since \texttt{symbol_map} has \(|\Sigma|\) entries, \(K\) linear passes 
may be costly. If an index of size \(2^L\) is created so that only the symbols in \texttt{symbol_map} 
need to be looked at, then the linear search can be eliminated. For each block, this index 
needs to be sorted at a constant cost of \(2^L \log_2 2^L\). So, the \texttt{RE-View} algorithm runs in 
\(O(2|S| + K(2^L \log_2 2^L) + K2^L)\)
6.3 Byte-Aligned Coding in Blocks

Algorithm 6.2: Algorithm for RE-View.

input : \( S \), length of symbols in \( P \), alignment \( L \) in bits
output : \( S' \)

1 \( S' \leftarrow \{\}; P_k \leftarrow 0; B_k \leftarrow 0; D_k \leftarrow 0; W_k \leftarrow 0; block_start \leftarrow 0 \)
2 initialise symbol_map
3 for \( i < |S| \) do
4 \( W_k \leftarrow W_k + |\alpha_i| \)
5 \( B_k \leftarrow B_k + 1 \)
6 if symbol_map\([\alpha_i]\) = 0 then
7 \hspace{10pt} symbol_map\([\alpha_i]\) \leftarrow 1
8 \hspace{10pt} D_k \leftarrow D_k + 1
9 \hspace{10pt} if \( D_k = 2^L \) then
10 \hspace{20pt} code \leftarrow 0; j \leftarrow 0
11 \hspace{20pt} while code < D_k do
12 \hspace{30pt} if symbol_map\([j]\) = 1 then
13 \hspace{40pt} write \( \alpha_j \) to prelude buffer
14 \hspace{40pt} symbol_map\([j]\) \leftarrow code
15 \hspace{40pt} code \leftarrow code + 1
16 \hspace{30pt} j \leftarrow j + 1
17 \hspace{20pt} P_k \leftarrow \text{size of prelude buffer}
18 \hspace{20pt} write \( P_k, B_k, D_k, W_k \) to \( S' \)
19 \hspace{20pt} write prelude buffer to \( S' \)
20 \hspace{20pt} j \leftarrow block_start
21 \hspace{20pt} for \( j < i \) do
22 \hspace{30pt} write symbol_map\([\alpha_j]\) to \( S' \)
23 \hspace{30pt} \( P_k \leftarrow 0; B_k \leftarrow 0; D_k \leftarrow 0; W_k \leftarrow 0; block_start \leftarrow i \)
24 \hspace{30pt} re-initialise symbol_map
25 \hspace{20pt} k \leftarrow k + 1
26 return \((S')\)

When a set \( C \) is searched for in the compressed sequence, the prelude of every block must be decoded. If at least one symbol in \( C \) exists in the current block \( k \), then the corresponding codewords are also decoded. Otherwise, the set of codewords is skipped using the value \( B_k \) in the header. As is demonstrated shortly, since the prelude is about 36.1% of the compressed file, a scan for a rare phrase involves processing at least one-third of the sequence. If the limited amount of decoding involved in processing every prelude cannot be accepted, then one solution is to create a higher-level index, similar to the GLIMPSE system [Manber and Wu, 1994]. The GLIMPSE retrieval system partitions
Coding Considerations

a file system into blocks. The index indicates, for every term in the file system, the blocks in which it occurs. Once the blocks of interest have been determined, each block is sequentially searched.

Several potential enhancements to Re-View are possible. Disk access can be reduced by physically separating the preludes and the codewords into different files. Then, disk access is concentrated on the prelude file. Another modification would build a top-level index on blocks so that the preludes of blocks which do not contain the symbol of interest are avoided entirely. Such an index would be kept in memory. Extending this last idea to an inverted index on symbols is also possible. None of these options were explored in the experiments reported in the next section since the search times obtained were acceptable.

6.4 Experiments

To investigate the compression effectiveness of the modifier streams and the reduced word sequence, experiments were conducted using Huffman coding, built on the Shuff implementation, Re-View, and an even simpler binary code. The efficiency for both skipping to a precise location and searching are also examined. Test files were taken from the punctuation-aligned Re-Pair experiments with method E of Re-Merge on the NEWS test data, as described in Chapter 5. The compression results from that chapter were presented in Table 5.9 (page 147) for the modifiers and Table 5.11 (page 148) for the reduced word sequence. In those tables, the modifiers and the reduced word sequence were coded with Shuff as a single block. The two tables in Chapter 5 also indicate that the combined compression ratio of the word lexicon, non-word lexicon, and the phrase hierarchy was 0.143 bits per character. The remaining 4 streams occupied 2,090 bits per character.

Additional information pertaining to the four streams is listed in Table 6.2. The last column in the table lists the self-entropy of each file. Note that because these values are relative to the streams themselves and not to the size of NEWS, they are not comparable to the compression ratios achieved in Table 5.9 and Table 5.11.
6.4 Experiments

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Symbol range</th>
<th>Distinct values</th>
<th>Number of symbols</th>
<th>Self-entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>case folding</td>
<td>0 to 65,536</td>
<td>634</td>
<td>168,721,097</td>
<td>1.303</td>
</tr>
<tr>
<td>stemming</td>
<td>0 to 7,864,320</td>
<td>541</td>
<td>168,721,097</td>
<td>2.204</td>
</tr>
<tr>
<td>non-word</td>
<td>0 to 11,332</td>
<td>11,333</td>
<td>168,721,097</td>
<td>2.054</td>
</tr>
<tr>
<td>reduced word</td>
<td>0 to 8,277,359</td>
<td>6,232,453</td>
<td>58,685,281</td>
<td>18.659</td>
</tr>
</tbody>
</table>

Table 6.2: Additional information about the modifier streams and the reduced word sequence from method E of RE-MERGE, with punctuation-aligned RE-PAIR on NEWS. The column headed self-information is the zero-order self-information for the frequency distribution of the symbols in that file, measured for each file in bits per symbol relative to the size of that file.

The other three columns of the table provide the range of symbols, the number of distinct values, and the total number of symbols of each stream. Note that the case folding and stemming mechanism employed by PRE-PAIR permit $2^{16}$ possible case folding values and $2^{23}$ possible stemming values. In practice it appears that even a large collection only has a small number of distinct values used. In contrast, PRE-PAIR assigns non-word modifier values in sequential order, starting from 1, with 0 reserved for the zero-length non-word symbol. For the non-word modifier stream, every symbol within the range appears. Since there is a one-to-one correspondence between a word token and its modifiers, the three modifier streams have the same number of values in them.

Compared to the other three streams, the reduced word sequence is shorter, because RE-PAIR and RE-MERGE have been applied. No pair of adjacent symbols occurs twice in this sequence, unless one of the restrictions imposed by punctuation-aligned RE-PAIR has been violated. As a result, the self-entropy of the stream is significantly higher than the others. Accordingly, the number of distinct values and the symbol range are also high. The symbol range is simply the number of symbols in the phrase hierarchy (see Table 5.10 on page 148), with a special end of block marker. After recursive pairing, some symbols in the phrase hierarchy do not appear directly in the reduced word sequence, but instead have been absorbed by higher generation symbols. That is why the number of distinct symbols is less than the number of entries in the phrase hierarchy.

There are two sets of experiments that are performed in this section. The first set considers the three modifier streams, while the second set examines the reduced word
Coding Considerations

sequence in isolation. In both scenarios, the compression ratios are reported, as are the times for compression and full decompression. The times for seek or search operations are also reported, whichever is applicable. When necessary, the compression ratios are reported in bits per character (bpc) relative to the size of NEWS, and in bits per symbol (bps) relative to the number of symbols in the file itself. The number of bits per character is comparable with all compression ratios reported in previous chapters, while the number of bits per symbol is important when examining the column labelled “self-entropy” in Table 6.2.

Program execution times in this chapter are reported as both CPU time and elapsed time. CPU time is a measure of the amount of computation performed by the program. All execution times reported in previous chapters have been CPU costs. But because this chapter also considers the amount of time a user must wait for a response, efficiency is also reported as elapsed time (also called “real time”). Elapsed time includes both user time and the time taken by the system for the program. Since all programs are executed with no data in memory beforehand, the time needed to access the disk is included in the elapsed time. All times, both CPU and elapsed, are averaged over three trials unless otherwise indicated.

Table 6.3 and Table 6.4 summarise the experiments with the modifiers. Each of the three modifier streams was compressed with indexed SHUFF with block sizes increasing by a factor of 8 from from 4KiB up to 16,384KiB. While not the intended purpose of RE-VIEW, RE-VIEWL was tested with $L = 8$ and $L = 16$ as a preview for the second set of experiments later. A flat binary coder (FLAT-BINARY) was also applied so that each symbol was encoded as a fixed-width binary code whose length was based on the maximum symbol value in the stream. The FLAT-BINARY system treated each stream as a single block.

Generally, as the block size for indexed SHUFF increased, compression effectiveness improved, as shown in Table 6.3. However, the decrease is small, especially between the larger block sizes. Also, the difference in compression ratio between the smallest block size and the largest block size shown is no more than 0.052 bits per character. Similar
### 6.4 Experiments

<table>
<thead>
<tr>
<th>Block size (× 1024 symbols)</th>
<th>Compression ratio bpc</th>
<th>bps of blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>case folding modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.259</td>
<td>1.608</td>
</tr>
<tr>
<td>32</td>
<td>0.243</td>
<td>1.508</td>
</tr>
<tr>
<td>256</td>
<td>0.240</td>
<td>1.494</td>
</tr>
<tr>
<td>2,048</td>
<td>0.240</td>
<td>1.492</td>
</tr>
<tr>
<td>16,384</td>
<td>0.240</td>
<td>1.493</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>2.736</td>
<td>17,000</td>
</tr>
<tr>
<td>Re-View</td>
<td>1.288</td>
<td>8.004</td>
</tr>
<tr>
<td>Re-View_16</td>
<td>2.575</td>
<td>16.004</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.429</td>
<td>2.667</td>
</tr>
<tr>
<td>32</td>
<td>0.388</td>
<td>2.414</td>
</tr>
<tr>
<td>256</td>
<td>0.379</td>
<td>2.358</td>
</tr>
<tr>
<td>2,048</td>
<td>0.378</td>
<td>2.347</td>
</tr>
<tr>
<td>16,384</td>
<td>0.377</td>
<td>2.346</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>3.701</td>
<td>23,000</td>
</tr>
<tr>
<td>Re-View</td>
<td>1.487</td>
<td>9.240</td>
</tr>
<tr>
<td>Re-View_16</td>
<td>2.642</td>
<td>16.417</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.370</td>
<td>2.298</td>
</tr>
<tr>
<td>32</td>
<td>0.346</td>
<td>2.148</td>
</tr>
<tr>
<td>256</td>
<td>0.342</td>
<td>2.123</td>
</tr>
<tr>
<td>2,048</td>
<td>0.341</td>
<td>2.119</td>
</tr>
<tr>
<td>16,384</td>
<td>0.342</td>
<td>2.126</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>2.253</td>
<td>14,000</td>
</tr>
<tr>
<td>Re-View</td>
<td>1.291</td>
<td>8.020</td>
</tr>
<tr>
<td>Re-View_16</td>
<td>2.575</td>
<td>16,004</td>
</tr>
</tbody>
</table>

Table 6.3: Compression effectiveness of indexed SUFF, Re-View, and FLAT-BINARY for the modifier streams. The second column lists the compression ratios of each method relative to the size of NEWS in bits per character relative to the original file. In the third column, compression levels in bits per symbol relative to the number of symbols in that stream are reported, with the best values for each stream highlighted. The number of blocks produced are shown in the last column.

Conclusions can be drawn from the column titled “bps” when compared with the self-entropy of each file (Table 6.2).

The implementation of Re-View first creates an initial buffer of 1,048,576 symbols, preventing any block from being larger than the buffer. The low variability in case folding modifier values means that after Re-View has seen just over one million symbols, less
Coding Considerations

than $2^8$ distinct symbols have been found. As the number of distinct values increases in first the stemming modifiers, and then the non-word modifiers, more blocks are made by RE-View$_8$. However, as RE-View$_{16}$ still only creates 161 blocks in both cases, no buffer has $2^{16}$ or more distinct values. Compression effectiveness of both types of RE-View are unacceptable, though quite expected. RE-View$_L$ can never achieve compression effectiveness that is better than $L$ bits per symbol.

The ideal method for encoding the modifier streams depends on efficiency as well as effectiveness, as shown in Table 6.4. The second and third columns of the table list the encoding and decoding times of each of the methods, averaged over three trials, and represented as CPU time. In order to demonstrate the random access abilities, 100 random positions were selected from the possible 168,721,007. During the design of the experiments, it was noted that access times varied depending on the position in the stream that is being accessed. However, if the same 100 positions were used by all of the experiments, then the access times between systems can be compared. Moreover, since the access times were short, Table 6.4 presents the total time for accessing all 100 positions in the same order. These times are reported as elapsed time and CPU time in the fourth and fifth columns of the table. All times have been averaged over three trials. In the entries labelled “—”, less than 0.1 seconds was noted, a time which cannot be reliably measured.

As less blocks are created, compression and decompression efficiency improves. RE-View is faster than both indexed SHUFF and FLAT-BINARY. The encoding times of FLAT-BINARY are equivalent to the largest block size tested for indexed SHUFF, while decoding is slightly worse than indexed SHUFF with a 256 KiB block size. The time required for random access is fastest for the FLAT-BINARY coder, followed by the two RE-View systems, and then indexed SHUFF in increasing block size.

The second set of experiments looked at encoding the reduced word sequence. Three methods of searching within the word sequence were compared,.

The first method applied version 2.4.2 of GNU GREP [Free Software Foundation, 2000] on a simplified version of NEWS. The GREP program searches for a set of one or more strings and returns the lines which match. Including GREP in the experiments allows

176
### 6.4 Experiments

<table>
<thead>
<tr>
<th>Block size (× 1024 symbols)</th>
<th>Compression</th>
<th>Random access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encoding time (CPU sec)</td>
<td>Decoding time (CPU sec)</td>
</tr>
<tr>
<td>case folding modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>270.6</td>
<td>444.5</td>
</tr>
<tr>
<td>32</td>
<td>47.1</td>
<td>64.8</td>
</tr>
<tr>
<td>256</td>
<td>24.4</td>
<td>18.7</td>
</tr>
<tr>
<td>2,048</td>
<td>20.7</td>
<td>12.7</td>
</tr>
<tr>
<td>16,384</td>
<td>19.7</td>
<td>11.2</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>15.3</td>
<td>28.9</td>
</tr>
<tr>
<td>Re-View8</td>
<td>8.3</td>
<td>3.3</td>
</tr>
<tr>
<td>Re-View16</td>
<td>8.7</td>
<td>3.5</td>
</tr>
<tr>
<td>stemming modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>916.5</td>
<td>445.1</td>
</tr>
<tr>
<td>32</td>
<td>141.1</td>
<td>66.0</td>
</tr>
<tr>
<td>256</td>
<td>36.8</td>
<td>19.1</td>
</tr>
<tr>
<td>2,048</td>
<td>23.1</td>
<td>12.4</td>
</tr>
<tr>
<td>16,384</td>
<td>21.0</td>
<td>11.3</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>17.0</td>
<td>37.6</td>
</tr>
<tr>
<td>Re-View8</td>
<td>9.0</td>
<td>4.1</td>
</tr>
<tr>
<td>Re-View16</td>
<td>8.7</td>
<td>4.0</td>
</tr>
<tr>
<td>non-word modifiers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>261.5</td>
<td>443.5</td>
</tr>
<tr>
<td>32</td>
<td>45.1</td>
<td>65.0</td>
</tr>
<tr>
<td>256</td>
<td>24.8</td>
<td>18.8</td>
</tr>
<tr>
<td>2,048</td>
<td>20.9</td>
<td>13.1</td>
</tr>
<tr>
<td>16,384</td>
<td>19.7</td>
<td>11.3</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>14.7</td>
<td>25.2</td>
</tr>
<tr>
<td>Re-View8</td>
<td>8.4</td>
<td>2.7</td>
</tr>
<tr>
<td>Re-View16</td>
<td>8.7</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 6.4: Compression efficiency of indexed shuff, Re-View, and flat-binary for the modifier streams of NEWS. All times are measured in seconds, with the best values shaded. Compression and decompression times are CPU times averaged over three trials. Results from randomly accessing 100 positions in each stream have also been repeated three times each. These times are shown as elapsed time and user time in the fourth and fifth columns, respectively. Times indicated by “—” are less than 0.1 seconds.

Comparisons between searching for a string and searching for a reference to a string in the phrase hierarchy. The NEWS document must be simplified in order to minimise the number of differences between these two tasks. The modified NEWS document contains
Coding Considerations

every symbol in the reduced word sequence expanded into word tokens. No modifiers are applied, and a single space is inserted between each word token. In order to allow GREP to return lines which match, linefeeds were inserted after each SGML closing tag. As long as no line that is searched contains a SGML closing tag, this change ensures that no search string is missed because it straddles multiple lines. One minor problem with this arrangement occurs when a search string appears twice or more in the same line. In this case, GREP simply returns one copy of the line, and does not continue searching the remainder. Hence, these modifications to the NEWS document are not perfect, but do provide a reasonable estimate for direct searching. The modified version of NEWS is 860 MiB in size.

While GREP is used to search for a particular string, the string may exist in several symbols in the phrase hierarchy. The second and third methods are applied to the reduced word sequence so that a set of symbols can be searched for simultaneously. The second method assumes the sequence is compressed with FLAT-BINARY or SHUFF (without an index) using various block sizes, with the word sequence fully decompressed for each search. The third method compresses the sequence using Re-VIEW$_{16}$ with searching performed using the partial-decoder approach described earlier in Section 6.3. Note that Re-VIEW$_{8}$ cannot be sensibly applied to the word sequence because of the large number of distinct symbols.

Table 6.5 presents compression and decompression results with SHUFF, Re-VIEW$_{16}$, and FLAT-BINARY. Larger block sizes were selected for SHUFF than for indexed SHUFF in order to favour compression effectiveness and decompression time. As Table 6.4 showed, the overhead of processing many small blocks with indexed SHUFF had an unfavourable effect on the decompression time. Block sizes for SHUFF ranged from 1,048,576 symbols per block up to the entire stream as a single block ($\infty$).

The compression ratios are listed in the second and third columns of the table in bits per character relative to the source text and bits per symbol relative to the file of phrase numbers, respectively. As the SHUFF block size increases, compression effectiveness improves. The exception occurs with the last block size. Enlarging the block size to $\infty$
Table 6.5: Compression statistics for the reduced word sequence of NEWS, using SHUFF, FLAT-BINARY, and RE-VIEW\textsubscript{16}. All times are reported in CPU seconds and averaged over three trials, with the best values shaded.

<table>
<thead>
<tr>
<th>Method</th>
<th>Compression ratio bpc</th>
<th>Compression ratio bps</th>
<th>Number of blocks</th>
<th>Compression time Encoding (s)</th>
<th>Decoding (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHUFF block size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,048,576</td>
<td>1.183</td>
<td>21.144</td>
<td>56</td>
<td>104.5</td>
<td>25.1</td>
</tr>
<tr>
<td>2,097,152</td>
<td>1.166</td>
<td>20.827</td>
<td>28</td>
<td>87.3</td>
<td>22.9</td>
</tr>
<tr>
<td>4,194,304</td>
<td>1.151</td>
<td>20.570</td>
<td>14</td>
<td>75.9</td>
<td>21.1</td>
</tr>
<tr>
<td>8,388,608</td>
<td>1.139</td>
<td>20.355</td>
<td>7</td>
<td>64.9</td>
<td>18.7</td>
</tr>
<tr>
<td>16,777,216</td>
<td>1.132</td>
<td>20.219</td>
<td>4</td>
<td>56.3</td>
<td>17.0</td>
</tr>
<tr>
<td>20,971,520</td>
<td>1.131</td>
<td>20.199</td>
<td>3</td>
<td>53.6</td>
<td>16.2</td>
</tr>
<tr>
<td>33,554,432</td>
<td>1.127</td>
<td>20.138</td>
<td>2</td>
<td>48.3</td>
<td>15.1</td>
</tr>
<tr>
<td>$\infty$</td>
<td>1.131</td>
<td>20.213</td>
<td>1</td>
<td>52.6</td>
<td>13.9</td>
</tr>
<tr>
<td>FLAT-BINARY</td>
<td>1.287</td>
<td>23,000</td>
<td>1</td>
<td>5.8</td>
<td>13.1</td>
</tr>
<tr>
<td>Re-View\textsubscript{16}</td>
<td>1.402</td>
<td>25,044</td>
<td>672</td>
<td>64.9</td>
<td>7.9</td>
</tr>
</tbody>
</table>

Reduces the total prelude size, but the coded message increases in size. The reason for this unexpected result is that the reduced word sequence, and the NEWS file from which it was derived, is not uniform. Recall that NEWS is composed of several years’ worth of news articles from two different sources, with the Wall Street Journal articles at the beginning and the Associated Press articles at the end. The change in document style and the change in article years was the price paid in order to create a sufficiently large document for experiments in this thesis. The FLAT-BINARY coder attains a compression level which is about 3 bits per symbol worse than SHUFF. RE-View\textsubscript{16} achieves compression which is 5 bits per symbol worse than SHUFF; not a surprising result given that RE-View\textsubscript{16} is not an entropy coder and a prelude must be transmitted for each of the 672 blocks. The size of the encoded reduced word sequence with RE-View\textsubscript{16} was 175.2 MiB, of which the values in the headers ($W_k$, $D_k$, $B_k$, and $P_k$) and the preludes made up 63.3 MiB. The large preludes ensure that RE-View\textsubscript{16} yields compression effectiveness which is also worse than FLAT-BINARY.

Following the column of block count are the encoding and decoding times, in seconds of CPU time, averaged over three trials. Generally, encoding time with SHUFF decreases as the block size increases, except for the “$\infty$” block size. Decoding time also decreases
Coding Considerations

for large block sizes as fewer preludes need to be decoded. The FLAT-BINARY coder is the fastest system for encoding and is faster than SHUFF for decoding, irrespective of block size. RE-VIEW_{16} requires as much time to encode as SHUFF with a block size of 8 million symbols. But RE-VIEW has the lowest decoding time out of all the systems tested.

More important than decoding time is searching time, which was the topic for this set of experiments. To quantify the searching speeds, each of GREP, decompressed searching, and RE-VIEW_{16} were used to search for symbols selected at random from the phrase hierarchy. Decompressed searching includes full decoding with SHUFF, followed by a direct search on the reduced word sequence. Since an infinite block size was employed by SHUFF, all the times for decompressed search include 16.0 seconds of elapsed time and 13.9 seconds of CPU time.

The symbols searched for were selected based on certain criteria. Only symbols which directly appeared in the reduced word sequence and were not components of any higher generation symbol in the phrase hierarchy were chosen. These restrictions ensure that the comparison was as fair as possible. Of the 8,277,359 symbols in the phrase hierarchy, 4,376,175 symbols were deemed valid.

Experiments were conducted from two different perspectives, based on the fact that not all of the valid symbols are equal. The first experiment highlights the fact that symbols occur with different frequencies. The four million valid symbols were divided into 3 equal groups based on frequency in the reduced word sequence: high, medium, and low. The second set of experiments was based on symbol length, in word tokens (or primitives). The valid symbols were divided into three groups based on their lengths when expanded, as shown in Table 6.6. Also shown in this table is the number of invalid symbols (second column). Valid symbols were divided based on their lengths and then merged to create three roughly equal symbol length groupings, as shown in the third and then the fourth columns of the table. In the first group, valid symbols of length 1 or 2 primitives are included. The second group contains valid symbols of lengths 3 or 4 primitives. All remaining valid symbols are in the third group.

Figure 6.3 and Table 6.7 present the results from the first set of experiments, based on
6.4 Experiments

<table>
<thead>
<tr>
<th>Symbol lengths (words)</th>
<th>Distribution</th>
<th>Size of group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Invalid candidates</td>
<td>Valid candidates</td>
</tr>
<tr>
<td>1</td>
<td>133,389</td>
<td>203,349</td>
</tr>
<tr>
<td>2</td>
<td>615,548</td>
<td>712,559</td>
</tr>
<tr>
<td>3</td>
<td>804,927</td>
<td>1,149,698</td>
</tr>
<tr>
<td>4</td>
<td>623,040</td>
<td>890,850</td>
</tr>
<tr>
<td>5</td>
<td>415,727</td>
<td>507,765</td>
</tr>
<tr>
<td>6</td>
<td>275,483</td>
<td>281,307</td>
</tr>
<tr>
<td>7</td>
<td>192,309</td>
<td>156,154</td>
</tr>
<tr>
<td>8</td>
<td>141,607</td>
<td>101,003</td>
</tr>
<tr>
<td>9</td>
<td>106,609</td>
<td>62,752</td>
</tr>
<tr>
<td>10+</td>
<td>592,544</td>
<td>310,738</td>
</tr>
<tr>
<td>Total</td>
<td>3,901,184</td>
<td>4,376,175</td>
</tr>
</tbody>
</table>

Table 6.6: Distribution of invalid and valid symbol for the experiments with the reduced word sequence. Phrase hierarchy symbols are separated based on length (in primitves), with the valid ones merged to form three symbol length groupings.

symbol frequency in the reduced word sequence. For each grouping, 1, 10, or 100 symbols were searched for, selected at random. The same symbols were searched three times and the time required was averaged, producing the elapsed and CPU times shown in the figure.

As more symbols are searched simultaneously, grep requires more time. However, the frequency of the symbols has a relatively small effect on the elapsed and CPU times. There is a noticeable difference between elapsed and CPU times for grep, due to the number of disk accesses. Meanwhile, decompressed searching requires the same amount of elapsed and CPU times, regardless of the number of symbols sought or the frequency of the symbols. Of the three systems shown in the figure, RE-VIEW16 achieves the best search times. As RE-VIEW16 searched for more symbols at the same time, more time is required. The effect of symbol frequency on search time begins to be noticeable when as many as 100 symbols are searched for. This result reflects the data displayed in the last two columns of Table 6.7. The third column indicates the number of blocks that were fully decoded. When searching with RE-VIEW16, every prelude must be decoded, but when the symbols being sought are not in the block, the rest of the block can be skipped. As a consequence of this property, the number of bytes read can never be less than the
Figure 6.3: Search times with GREP, decompressed search, and RE-VIEW_16 for the reduced word sequence based on frequency groupings. The elapsed times and CPU times are reported in seconds and averaged over three trials with the same test symbols.

combined size of all the preludes, which is 63.3 MiB. And, as the fourth column shows, as more blocks need to be fully decoded, more bytes need to be read. So, when either more symbols or more frequent symbols are searched for, the number of blocks that are fully decoded and the number of bytes read increases accordingly.
### 6.4 Experiments

<table>
<thead>
<tr>
<th>Frequency group</th>
<th>Symbols searched</th>
<th>Number of blocks decoded</th>
<th>Number of bytes read</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>1</td>
<td>4</td>
<td>63.9 MiB</td>
</tr>
<tr>
<td>High</td>
<td>10</td>
<td>58</td>
<td>73.0 MiB</td>
</tr>
<tr>
<td>High</td>
<td>100</td>
<td>423</td>
<td>134.0 MiB</td>
</tr>
<tr>
<td>Medium</td>
<td>1</td>
<td>2</td>
<td>63.6 MiB</td>
</tr>
<tr>
<td>Medium</td>
<td>10</td>
<td>18</td>
<td>66.3 MiB</td>
</tr>
<tr>
<td>Medium</td>
<td>100</td>
<td>161</td>
<td>90.0 MiB</td>
</tr>
<tr>
<td>Low</td>
<td>1</td>
<td>1</td>
<td>63.4 MiB</td>
</tr>
<tr>
<td>Low</td>
<td>10</td>
<td>14</td>
<td>65.6 MiB</td>
</tr>
<tr>
<td>Low</td>
<td>100</td>
<td>144</td>
<td>87.4 MiB</td>
</tr>
</tbody>
</table>

Table 6.7: Search efficiency with Re-View\textsubscript{16} for the reduced word sequence based on frequency groupings. The number of blocks fully decoded and the number of bytes read in order to complete the searching are shown in the third and fourth columns.

The results from the second set of experiments with the reduced word sequence are contained in Figure 6.4 and Table 6.8. If Figure 6.3(c) is compared with Figure 6.4(c), it would appear that grep requires less time as symbols which expand to more words are sought. However, symbol length does not have a noticeable effect on decompressed searching or Re-View\textsubscript{16}, according to these graphs.

Table 6.8 shows how less blocks are requested as symbol length increases. But this trend may be caused by the fact that symbols with more words are expected to occur less frequently.

In conclusion, this section has looked at the coding requirements of the three modifier streams and the reduced word sequence. Indexed shuff produced the best compression effectiveness when sufficiently large block sizes were chosen. While the compression and random access times offered by Re-View and flat-binary for the modifiers were good, indexed shuff with small block sizes gave comparable results for skipping to arbitrary locations within the streams.

Three compression methods were examined for the reduced word sequence. As with the modifier streams, shuff provided the best compression effectiveness. While the flat-binary coder also gave better compression levels than Re-View\textsubscript{16}, Re-View\textsubscript{16} was found
Coding Considerations

![Graphs showing search times with GREP, decompressed search, and Re-View16 for symbol length groupings.](image)

Figure 6.4: Search times with GREP, decompressed search, and Re-View16 for the reduced word sequence based on symbol length groupings. The elapsed times and CPU times are reported in seconds and averaged over three trials with the same test symbols.

to be more efficient than SHUFF and FLAT-BINARY for decompression and searching. In these experiments, a modified version of NEWS was used by GREP, while decompressed searching assumed SHUFF with an infinite block size. The times for decompressed searching and Re-View16 were not affected by symbol frequency or symbol length, but both
6.5 Selecting Coders

<table>
<thead>
<tr>
<th>Symbol length group</th>
<th>Symbols searched</th>
<th>Number of blocks decoded</th>
<th>Number of bytes read</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 - 2</td>
<td>1</td>
<td>2</td>
<td>63.6 MiB</td>
</tr>
<tr>
<td>1 - 2</td>
<td>10</td>
<td>41</td>
<td>70.2 MiB</td>
</tr>
<tr>
<td>1 - 2</td>
<td>100</td>
<td>298</td>
<td>113.0 MiB</td>
</tr>
<tr>
<td>3 - 4</td>
<td>1</td>
<td>2</td>
<td>63.6 MiB</td>
</tr>
<tr>
<td>3 - 4</td>
<td>10</td>
<td>30</td>
<td>68.3 MiB</td>
</tr>
<tr>
<td>3 - 4</td>
<td>100</td>
<td>261</td>
<td>107.0 MiB</td>
</tr>
<tr>
<td>5+</td>
<td>1</td>
<td>2</td>
<td>63.6 MiB</td>
</tr>
<tr>
<td>5+</td>
<td>10</td>
<td>32</td>
<td>68.6 MiB</td>
</tr>
<tr>
<td>5+</td>
<td>100</td>
<td>181</td>
<td>93.7 MiB</td>
</tr>
</tbody>
</table>

Table 6.8: Search efficiency with Re-View16 for the reduced word sequence based on symbol length groupings. The number of blocks fully decoded and the number of bytes read in order to complete the searching are shown in the third and fourth columns.

conditions had a noticeable effect on the number of blocks fully decoded and the number of bytes read from disk by Re-View16. Furthermore, as more symbols are searched for simultaneously, both Re-View16 and grep required more time.

6.5 Selecting Coders

This chapter has examined coders for the three modifier streams and the reduced word sequence stream. The decoding requirements of these four streams for phrase browsing are quite different from those that apply to the phrase hierarchy and the two lexicons. Within these four streams, different decoding specifications also emerge. The modifier streams require random access, whereas the reduced word sequence is searched.

The naive approach to coding these four streams is to simply encode each reference as a 32-bit integer. Retrieval time is fast for both random access and searching, and the difficulty is allocating enough storage. Referring to Table 6.2, such an approach would require 4,310 bits per character for only these streams. If this level of compression is acceptable, then such a coding scheme can be considered.

At the other extreme, a compression system such as BZIP2 can be applied to the modifier streams, as explored in Chapter 4. If such an approach was employed, then the
Coding Considerations

streams must be fully decoded before being used, yielding unfavourable response times.

As a balance between these extremes, indexed Huffman coding is a reasonable choice for the modifier streams. As the streams are required less frequently, if blocks are created with an index for each block, then at most two blocks are decoded at any one time for seeking. The ideal random access times occur when a block size of 32,768 symbols is selected. If this block size is applied, the combined compression ratio of the three modifier streams is 0.977 bits per character.

The reduced word sequence must be searchable and Re-View\textsubscript{16} combines byte-alignment for the codewords with the ability to skip blocks when it has been determined that no symbol of interest occurs within the block. The disadvantages of this approach are that all of the preludes must be decoded as part of every search, and that compression effectiveness is not as good as if an entropy coder like SHUFF had been used. One improvement to Re-View\textsubscript{16}, which was not explored, would be to create a second-level index so that each token in the file is associated directly with a list of the blocks in which it occurs.

Re-View\textsubscript{16} achieves better search times than GREP and decompressed searching. But the GREP system still performs remarkably well given that the search criteria were strings instead of sets of symbols. Compression of the reduced sequence with Re-View\textsubscript{16} requires 1.402 bits per character.

So, through the techniques described in this chapter and the encoding methods decided earlier for the phrase hierarchy and lexicons, the compression effectiveness for all seven streams is 2,523 bits per character. Table 6.9 updates Table 4.16 on page 106 with the coding mechanisms examined in this chapter. The next chapter takes these files and shows how they can be used to construct a phrase browsing tool.
### 6.5 Selecting Coders

<table>
<thead>
<tr>
<th>Output from PRE-PAIR</th>
<th>word lexicon</th>
<th>word sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>phrase browsing</td>
<td>searching for symbols</td>
</tr>
<tr>
<td>Model</td>
<td>front coding and BZIP2</td>
<td>RE-PAIR</td>
</tr>
<tr>
<td>Coder</td>
<td>phrase hierarchy</td>
<td>reduced word sequence</td>
</tr>
<tr>
<td></td>
<td>chiastic slide and interpolative coding</td>
<td>RE-VIEW16</td>
</tr>
</tbody>
</table>

(a) Words

<table>
<thead>
<tr>
<th>Output from PRE-PAIR</th>
<th>non-word lexicon</th>
<th>non-word modifiers</th>
<th>case folding modifiers</th>
<th>stemming modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td></td>
<td>skipping to symbols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>front coding and BZIP2</td>
<td>indexed SHUFF</td>
<td>indexed SHUFF</td>
<td>indexed SHUFF</td>
</tr>
<tr>
<td>Coder</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Non-words and modifiers

Table 6.9: The seven streams produced by PRE-PAIR and RE-PAIR. The purpose of each stream, and the compression methods employed are taken from Table 4.16 on page 106, updated with the coding mechanisms in this chapter.
Chapter 7

Phrase Browsing

The main focus of the previous chapters has been the problem of transforming a message into a more economical form which is also suitable for phrase browsing. Decisions had to be made with respect to the level of word-based pre-processing necessary and the efficiency of coders. All of these issues were considered in the context of the sender of the compressed message. In this chapter, that compressed message is viewed from the point of view of its receiver, in the form of a phrase browser.

The phrase browser is a tool which hides the complexity and size of the compressed message from the user. The browser described in this chapter not only allows a user to gain an understanding of a document through its phrases, but it also provides a searching facility. As alluded to in Chapter 6, after phrase browsing ends and a phrase of interest has been found, the contexts in which the phrase appears may be needed. The phrase browser searches for these locations and also allows the user to refine the search result by incorporating case, stemming, and non-word information. Continuing with the naming scheme adopted by this thesis, the phrase browser is called RE-PHINE, in that both the phrase browsing and the subsequent searching are opportunities for the user to continually refine the initial query.

The remainder of this chapter is structured as follows. The steps embodied by the phrase browser are further elaborated in Section 7.1. In order to efficiently browse the phrase hierarchy, a graph data structure is required. This data structure is described in Section 7.2. A demonstration of RE-PHINE is given in Section 7.3. Then, RE-PHINE is
Phrase Browsing

compared and contrasted with related work and traditional IR techniques in Section 7.4, including an assessment of the quality of phrases made available to the user.

7.1 Browsing and Searching Steps

The RE-PHINE system encompasses both phrase browsing and context searching. The precise steps that are available to the user in order to achieve these aims depend heavily on the seven streams produced during the compression process. Recall that the seven streams were the phrase hierarchy, the word lexicon, the non-word lexicon, the reduced word sequence, the case folding modifiers, the stemming modifiers, and the non-word modifiers.

Implementation details about RE-PHINE are covered in the next section. In this section, only a general overview of RE-PHINE is provided, starting with Figure 7.1. Each rectangular box represents an action performed within RE-PHINE. Boxes with dashed borders are actions performed by RE-PHINE, whereas solid boxes represent actions performed by the system in direct response to the user's request. A shaded cylinder represents one of the seven streams available to RE-PHINE, with dashed lines symbolising file access. The sequence of steps commences at the top left-hand corner of the figure. An explicit end to the sequence of steps is not shown because the user can stop phrase browsing at any time.

In the first step of the figure, RE-PHINE decodes the phrase hierarchy, the word lexicon, and the non-word lexicon, and maintains all three within memory. Next, a directed graph (DG) is built using the symbols of the phrase hierarchy, to permit efficient searching. Details about the structure of the graph is given in the next section. The third step is interactive phrase browsing. In browsing the phrase hierarchy, the user interactively traverses the DG until a symbol of interest, \( \alpha \) say, has been isolated.

In the fourth step, the symbol of interest is used to identify all other symbols in the DG which contain \( \alpha \). When that search completes, a set of symbols \( \mathcal{C} \) has been established. Next, \( \mathcal{C} \) is input to the fifth step, in which the RE-\textsc{View}_{16} decoder is used to locate every context in which any symbol in \( \mathcal{C} \) occurs. The RE-\textsc{View}_{16} decoder returns a result set, \( \mathcal{R} \). Suppose a symbol \( \beta \) contains the symbol \( \alpha \) as a component. In order to avoid reporting
7.1 Browsing and Searching Steps

Figure 7.1: The sequence of steps performed by RE-PHINE during phrase browsing. Dashed boxes indicate actions performed by the system, while solid boxes require input from the user. Shaded cylinders represent one of the seven streams accessible by RE-PHINE, with dashed lines from them representing file access. The user can exit the system at any time.
Phrase Browsing

the same location multiple times, when an occurrence of $\beta$ has been found in the reduced word sequence, only $\beta$ is reported. The size of $\mathcal{R}$ is denoted as $|\mathcal{R}|$, and members are enumerated from $i = 1$ to $i = |\mathcal{R}|$.

The first result’s context, $\mathcal{R}_1$, is displayed to the user. At this point, the user may choose any combination of five actions. These actions are: return to phrase browsing; toggle case folding modifiers, stemming modifiers, or non-word insertion for $\mathcal{R}_1$; or examine the next or previous result. It is assumed that the result set is a circular list, so that the previous result to $\mathcal{R}_1$ is $\mathcal{R}_{|\mathcal{R}|}$. The three actions related to the modifiers initiate seek operations to the corresponding positions in their respective compressed streams. After the modifiers have been retrieved, $\mathcal{R}_i$ is re-displayed to the user. The application of modifiers can be toggled in any order. The first time modifiers are required, access to the streams are performed. Assuming requested modifiers are buffered in memory, subsequent requests do not need to access the file again. Finally, if the contexts that were obtained were inadequate in satisfying the user’s need, any part of a context can be used as input into the phrase browsing part of RE-PHINE.

Later in this chapter, the entire phrase browsing process is detailed through a brief example which demonstrates the choices available to the user. But first, in the next section, the directed graph required for phrase browsing is explained.

7.2 Phrase Hierarchy as a Directed Graph

Once the phrase hierarchy has been decoded, a directed graph (DG) is constructed. Every symbol in the phrase hierarchy is assigned to a separate node in the DG. Each node contains six directed edges connecting it to other nodes, as shown in Figure 7.2. In this figure, the symbols are based on character-based RE-PAIR instead of punctuation-aligned RE-PAIR in order to simplify the discussion. The current node is represented by the symbol “w|o”, with the vertical bar (|) separating the symbol into its two components. Two of the six pointers are child pointers, and refer to a symbol’s structure These pointers were described in Chapter 5 as part of phase 1 of RE-MERGE, but were called components at that time. The remaining four pointers are collectively called navigational pointers.
7.2 Phrase Hierarchy as a Directed Graph

![Directed Graph Diagram]

Figure 7.2: The six pointers used by Re-Phine to link nodes representing symbols in the phrase hierarchy into a directed graph. The symbols in the example are based on character-based Re-Pair, for illustrative purposes only. The node of interest is in the centre, with the symbol “w|o”. A vertical bar (|) separates the expanded symbols into left and right components. Names of each pair of pointers are provided to the right of the figure.

Navigational pointers are added for the purpose of browsing and have not been required in the previous discussion.

The purpose of each pointer is illustrated in the figure. Pointers appear in pairs and are designated as being “left” or “right”. The four navigational pointers are in two pairs – the sibling pointers, and the parent pointers.

In the example figure, the current symbol is in the centre, with its six pointers listed in full. All of the other nodes in the graph have six pointers each as well, but the outgoing edges are omitted in the figure for simplicity. The centre symbol is “w|o”, with left and right components (or children) of “w” and “o”, respectively. The left sibling symbol of “w|o” is another symbol which also has “w” as a left child. A chain is formed from the centre node, with all symbols with “w” as a left child being linked together with their left sibling pointers. Within a chain, the nodes are unsorted, so any node can serve as the front of the list. Each chain ends with a final empty node. The child pointers of all primitives also lead to empty nodes. Similarly, the right sibling pointer from “w|o”, leads to another symbol with “o” as a right child.

One important operation is to obtain the entire chain when the current symbol is any
Phrase Browsing

Figure 7.3: The directed graph structure created by Re-Phine to link symbols together, at a larger scale than Figure 7.2. The node of interest is “w|o” in the centre. The dashed arrows show how an entire chain of symbols can be obtained from an arbitrary symbol \( \alpha \) within the chain. Also shown are two siblings of the symbol “wo|od”.

symbol along that chain. The child and parent pointers help solve this problem. Parent pointers directly lead to symbols which are a single extension away from the current symbol. An extension occurs when a symbol is juxtaposed to the current symbol on either side, arriving at another symbol in the phrase hierarchy. For example, in Figure 7.2, the right parent pointer from “w|o” allows a right extension to the symbol “wo|od”. Figure 7.3 shows Figure 7.2 at a larger scale to demonstrate how an entire chain of symbols can be obtained from anywhere in the list. The “w|o” is at the head of the list of right siblings, and the current symbol is \( \alpha \). Both of these symbols have the same right component. Every symbol along that chain has a right child link to the primitive “o”. The front of the chain can be reached by taking the right child pointer from \( \alpha \), followed by the left parent link of “o”, as shown by the dashed lines in the figure. This is possible provided that graph construction ensures that every node’s parent pointer leads to the beginning of a chain of parents. If a symbol does not have a parent, the parent is represented as an empty node.

The graph structure created by Re-Phine resembles the data structure employed by MK10 [Wolff, 1975] for the recursive pairing of symbols for discovering sentence structure. Each symbol in the dictionary created by MK10 has major and minor links. A major forwards link is the same as a left child pointer in Re-Phine’s graph, while a major

194
7.2 Phrase Hierarchy as a Directed Graph

backwards link is identical to a right child pointer. Minor links operate in a similar fashion to the sibling pointers of RE-PHINE.

Equipped with this graph, five operations are available to the user. Assuming the user’s currently selected symbol is “w|o” of Figure 7.2 and Figure 7.3, the five operations are as follows. First, the user can decompose the symbol into its two components through the child pointers. Second, if the right parent pointer from “w|o” is used, followed by all left sibling pointers until the empty node is reached, then all right extensions of “w|o” can be retrieved. In Figure 7.3, this operation would obtain the set of two symbols, {“wo|od”, “wo|uld”}. Likewise, a third operation is to extend “w|o” to the left. The union of these last two sets results in a fourth function, extending “w|o” in both directions.

The fifth and final operation is reserved for decoding, instead of phrase browsing. Decoding requires identification of all symbols which contain the current symbol $\alpha$, either directly or indirectly. The located symbols are denoted in Figure 7.1 as $C$. In order to find all symbols in the phrase hierarchy graph which contain $\alpha$, a breadth-first search with the navigational pointers is sufficient. Let $\beta$ represent a symbol which is an ancestor of $\alpha$. If $\beta$ has only one occurrence of $\alpha$, then all of its parents does as well. But $\alpha$ must appear in either its left or its right component, which can be determined based on how $\beta$ was reached. So, only one of the sibling pointers need to be followed. If $\beta$ has multiple occurrences of $\alpha$, with at least one in each component, then during the breadth-first search, $\beta$ is reached twice. Care must be taken to ensure that when $\beta$ is reached the second time, only the other sibling pointer is traversed, and not the parent pointers. If this strategy is followed, then the nature of the navigational pointers ensures that all symbols which include “w|o” are included.

Construction of the DG requires $O(|\Sigma| + |\rho|)$ time, for a phrase hierarchy of $|\Sigma|$ primitives and $|\rho|$ phrases. Since the chiastic slide and interpolative coding have been used to code the phrase hierarchy, the phrase hierarchy is already in increasing generation order, and within each generation, in increasing chiastic slide value order. As no symbol is in the same generation as either of its two components, constructing the graph from the first primitive symbol in sequential order results in graph construction resembling a bottom-
Phrase Browsing

up approach. If the phrase hierarchy is stored in memory as an array, and if each of the six graph pointers are represented as array indexes, then locating a symbol's components requires \(O(1)\) time. For each phrase, an insertion is performed into the appropriate parent pointers of each of its two components, so that each node is added at the front of its two chains of siblings in \(O(1)\) time. That is, a single linear pass through the phrase hierarchy is sufficient to construct the graph.

The memory space required for the phrase hierarchy is linear in the number of symbols in the phrase hierarchy. The graph structure adds an additional 6 words of memory per symbol. Also, the phrase hierarchy may have every symbol expanded into primitives, or symbol expansion may be deferred until the symbol is required. The choice made depends which is more important: memory space or response time. Experiments demonstrating the advantages and disadvantages of phrase hierarchy expansion were reported in Table 3.2 on page 49. In the implementation of RE-PHINE described in this chapter, a buffer of recently decoded symbols is maintained and consulted before decoding a symbol. This approach was also employed in RE-PAIR’s decoder, DES-PAIR.

7.3 Phrase Browsing Example

The RE-PHINE system performs two main tasks: phrase browsing and decoding symbol contexts for presentation. Examples of both tasks are presented in Figures 7.4 and 7.5. The aspects of RE-PHINE related to the graphical user interface were implemented with version 2.0.2 of the GTK+ library\(^1\).

The data for the example is NEWS, processed with punctuation-aligned RE-PAIR exactly as was described in the experiments of Chapter 5. The sequence of word tokens in those experiments were partitioned into blocks of 20,971,520 symbols each, and then merged using method E of RE-MERGE (Section 5.6). Modifier and sequence coding were done with indexed SHUFF and RE-VIEW\(_{16}\) respectively.

In the previous section, the choices available to the user for phrase browsing from

the current symbol were discussed. The user also needs to be given a starting point to browse from. In Re-Phine, the user is first presented with the document’s entire lexicon, as shown in Figure 7.4(a). The extensive list can be daunting, and one solution is to filter words, discussed later in the chapter. In Figure 7.4(a), the user has selected the word “prime”. The first symbol is extended to the right in Figure 7.4(b), and the phrase “prime minist” has been selected. While phrases in this window are ordered according to their locations in the DG, Jones and Paynter [1999] and Paynter et al. [2000] describe systems where phrases can be ordered according to the number of documents in which they appear, or their frequency in the collection. While details about the implementation of these two features are not available, it is expected that either the additional information is stored by the system, or calculated at run-time. Also note the variations on the phrase “prime minist” due to the different spellings of the second word. While stemming and case folding have reduced the size of the phrase hierarchy, it is still possible that two seemingly equivalent words stem differently. The chosen phrase is extended to the left and the phrase “canadian prime minist” has been highlighted in Figure 7.4(c). As this window shows, unfortunately, the words “Canadian” and “Canada’s” stem to different words. The identified phrase of Figure 7.4(c) is extended in both directions, resulting in Figure 7.4(d) and its 7 phrases. Of these 7 phrases, 3 are right extensions and 4 are left extensions to the phrase “canadian prime minist”. The selected phrase is broken down into its two components through its child pointers, as shown in Figure 7.4(e). This phrase browsing example commenced with the word “prime” and, in four steps, arrived at the name of a Canadian prime minister. In Re-Phine, the refinement of a query term is done interactively with the help of the phrase hierarchy’s graph.

A sample decoding session is illustrated in Figure 7.5. The selected phrase of Figure 7.4(e) is the starting point for the decoding. The first of 17 occurrences of the phrase “pierre elliott trudeau” is given in Figure 7.5(a). The results are presented to the user in the order in which they occur in the original document. In the first window, the phrase exists in a longer phrase, which has been marked. A fixed context of 10 symbols in either direction of the highlighted symbol has been selected for Re-Phine. In the absence of
Figure 7.4: A sample phrase browsing session of the NEWS document.
7.3 Phrase Browsing Example

detailed information about the non-words, a space character is inserted between each word token, with linefeeds added as appropriate. Even though the words in this window have been stemmed and case folded, reasonable guesses can be made with most of the words. In Figure 7.5(b), the user has skipped ahead to the 15th result. As this section of text may be of interest, the case folding is first reversed in Figure 7.5(c), and then the stemming modifiers are applied in Figure 7.5(d). The corresponding modifiers are located by the system and only the small number of required modifiers are transmitted to the RE-PHONE window. Modifiers can be toggled in any order, and allow the user to progressively change the displayed context. Finally, in Figure 7.5(e), the non-words are inserted into the text and the window is enlarged for improved display. The text exists in the original NEWS document as shown.

At this stage, the user's information need may be satisfied. Alternatively, the user may decide to select the phrase “Progressive Conservative” as the new starting point for phrase browsing. Then, the starting point for the browsing session becomes the phrase “progress conserv”.

RE-PHONE applies the modifiers only when requested by the user. This approach takes advantage of the fact that the modifier streams are encoded separately. At least two other ways of handling modifiers are possible, which have not been implemented for RE-PHONE. If the cost of decoding and transmitting the modifiers is acceptable, the user can be given the option to reverse all modifiers by clicking one button. This feature can be added by making a small change to the RE-PHONE interface. Another option is to display initial word tokens with default stems and case information applied. That is, the most frequent stemming and case folding modifier is recorded for each word in the lexicon during pre-processing with PRE-PAIR. The additional disk space required is 2|Σ| integers, uncompressed, for a compressed document collection of |Σ| primitives. When phrases are first displayed (for example, in Figure 7.5(a)), these default modifiers can be applied immediately without any intervention from the user. While the phrases presented may occasionally be linguistically incorrect, they should be valid most of the time. These default modifiers can be transmitted as needed, or once at the beginning of
Figure 7.5: A sample decoding session of the NEWS document, continuing from Figure 7.4(e).
the browsing session. A similar idea was adopted by Paynter et al. [2000]. The aim of both of these alternatives is to make the interface more user-friendly by minimising the amount of exposure users have to case folded, stemmed words.

7.4 Evaluating Re-Phine

Information retrieval systems are generally evaluated in terms of efficiency and effectiveness. Efficiency is measured as the amount of time required to return a response to the user. Effectiveness is an assessment of the extent to which the set of results satisfies the user’s information need.

Suppose an IR system indexes the words in a collection of \( n \) documents, and that for some set of queries, the number of documents that are relevant is known. The effectiveness of the IR system for the query is determined by the number of documents the system returns which are relevant. Two standard methods of quantifying effectiveness are precision and recall. These and other methods of judging retrieval effectiveness are described by various sources, including Korfhage [1997], Baeza-Yates and Ribeiro-Neto [1999], and Witten et al. [1999].

Precision and recall are expressed on a scale from 0 to 1. Precision is the fraction of returned results that are relevant, while recall is the proportion of all relevant documents that are returned. If a user poses a query for which the set of relevant documents in the collection is \( r \), and the set of documents returned by the system is \( a \), then precision and recall are defined as follows:

\[
\text{Precision} = \frac{|r \cap a|}{|a|} \quad (7.1)
\]

\[
\text{Recall} = \frac{|r \cap a|}{|r|} \quad (7.2)
\]

Ideally, the effectiveness of Re-Phine should be evaluated in a similar manner. However, significant differences between Re-Phine and traditional IR systems prevent any straightforward comparison.
Phrase Browsing

![Diagram of noun phrase structure]

Figure 7.6: Definition of a noun phrase, adapted from Greenbaum [1996, pg. 209].

Instead, RE-PHINE is evaluated in three steps. In the first step, an overview of past work related to phrase browsing is summarised. Second, RE-PHINE is analysed in detail in the context of related work and traditional query methods. Some possible improvements to RE-PHINE are also proposed. Finally, the phrase hierarchy used by RE-PHINE is compared to noun phrases formed through natural language processing techniques.

Related Work

Previous work in the area of phrase browsing can be classified based on the method employed for drawing phrases from a document. Phrases can be obtained through the semantic meaning of the words using natural language processing techniques, or statistically, by applying a mechanism similar to RE-PAIR.

Most work with natural language processing (NLP) techniques are related to tagging words according to their parts-of-speech (that is, noun, verb, etc.) and forming phrases according to the rules of grammar. A significant amount of work has concentrated on noun phrases. A noun phrase is a linguistic unit. In it, a noun is both required and designated as the most semantically important word and, as a result, is assigned as the phrase’s head. Figure 7.6 shows an example of a noun phrase, divided into its various parts.

While some retrieval systems treat a query as being made up of individual words, occasionally, a query should be seen as a set of one or more noun phrases. If a user posed the query “University of Melbourne”, ideally, documents about other universities or the city of Melbourne are not of interest. In fact, the seemingly unimportant word “of” is crucial for this phrase, since “University in Melbourne” has a completely different meaning. Kirsch [1998] provided some insight into the Internet search engine Infoseek and
noted that most users submitted queries which were noun phrases and that queries were, on average, 2.2 words in length.

Several researchers have made use of noun phrases for their phrase-based IR systems. The CLARIT system [Evans and Zhai, 1996] identified both simplex and complex noun phrases for the purpose of improved precision and recall in information retrieval, without the need for training data. The difference between a simplex noun phrase and a complex noun phrase is that a simplex noun phrase omits all prepositional phrases (the postmodifier in Figure 7.6). Anick and Vaithyanathan [1997] developed the PARAPHRASE system which partitioned documents into clusters and summarised each cluster with noun phrases for the purpose of browsing. A variant of the Brill tagger [Brill, 1994] labelled the words according to their parts-of-speech. LINKIT by Wacholder et al. [2001] identified noun phrases, which were subsequently displayed by the INTELL-INDEX interface. A part-of-speech tagger was used to determine simplex noun phrases, which were then grouped according to their heads. As a result, phrases such as “the university bookstore” was associated with the phrase “a bookstore”.

Extraction of keyphrases (document keywords in the form of phrases) with machine learning techniques was investigated by Frank et al. [1999] and embedded within a system dubbed KEA. Several interfaces have been built on top of KEA, including KNILES, PHRASIER, and KEYPHIND. KNILES and PHRASIER uses the phrases identified by KEA to link documents together via hyperlinks [Jones, 1999, Jones and Paynter, 1999]. Browsing phrases identified by KEA are supported by the KEYPHIND system [Gutwin et al., 1999]. KEYPHIND removes all but twelve phrases for each document in a collection and constructs three types of indexes with the phrases: a word-to-phrase index, a phrase-to-document index, and a document-to-phrase index. A user study was conducted which compared the usability of KEYPHIND to a traditional IR system based on ranked queries. The results of the study showed that KEYPHIND was preferred for certain browsing tasks, such as collection evaluation. A more recent user study [Jones and Paynter, 2001] with KEA and keyphrases looked at computer science papers and compared the phrases selected by KEA with those specified by the authors of each paper. The study’s conclusion was that the
Phrase Browsing

phrases selected by KEA were almost as good as ones chosen by the authors.

Phrase selection through statistical methods is more relevant to RE-PHINE than through semantic methods. The PHIND system of Nevill-Manning et al. [1997] selected phrases using the SEQUITUR compression mechanism, which is similar to RE-PAIR and was discussed in detail in Chapter 3. Like word-based RE-PAIR described here, SEQUITUR was modified to operate on word tokens in order to ensure word alignment. Punctuation marks were removed and words were case folded to lower case. Word stems were handled by the phrase browser so that if the word “book” was chosen, a search for the word with different endings was performed. Follow-up work on a browsing system built on top of SEQUITUR was described by Nevill-Manning et al. [1999]. In order to browse large document collections, a disk-based browsing system is proposed. The phrase hierarchy is placed on disk with an inverted index. Smaller auxiliary data structures are kept in main memory for accessing the phrase hierarchy.

Williams et al. [1999] describe an index structure called a nextword index which provides explicit support for phrase querying. This multi-level data structure is an index to pairs of words that appear in the document collection. The word lexicon forms the top-level of the index. Each word in the lexicon has a nextword list that, together, indicates the word pairs that exist in the collection. Every word pair has a position list as the third and final level in the nextword index. The position list specifies the documents and the document positions where the pairs of words occur. Subsequent extensions to this work include compaction techniques for improving storage requirements [Bahle et al., 2001a], and a description of how nextword indexes can be used for mono-directional phrase browsing [Bahle et al., 2001b]. The nextword index for a transformed version of WSJ508 occupied 56% of the collection size. When a query phrase with more than two words is processed, it must be partitioned into pairs of words for look-up into the index. The position lists obtained using the word pairs need to be combined in order to find locations of the original, longer phrase. Bahle et al. [2001b, 2002] showed how the word pairs should be selected in order to optimise querying time, and then implemented a retrieval system which combined a nextword index with a word-level IR system. It was shown that the
addition of the nextword index provided more efficient retrieval when the first word is a common word.

Several methods combine semantics with statistics for phrase selection. Wolff [1980] tagged words according to their parts-of-speech, and replaced frequently occurring pairs of word classifications with a recursive pairing mechanism similar to RE-PAIR. Paynter et al. [2000] improved on PHIND by combining SEQUITUR with KEA. A Brill tagger was combined with KEA and applied to the documents in order to identify boundaries between noun phrases. Next, SEQUITUR was applied to combine words into phrases, with constraints enforced to prevent any crossing over the boundaries.

A separate, yet related area to retrieval with phrases is that of locality-based indexing (or proximity indexing [Baeza-Yates and Ribeiro-Neto, 1999, pg. 101]). Locality-based indexing creates an index at the granularity of individual word tokens instead of documents. When multiple terms are included in the query, the results are chosen not only because of query frequency, but also the distance between query terms within the documents. Some past works with locality-based retrieval include those of Buckley et al. [1995], Clarke et al. [1995], Hawking and Thistlewaite [1995], and de Kretser and Moffat [1999].

**Analysing Re-Phine**

The RE-PHINE system of this chapter can be analysed in terms of its efficiency, effectiveness, and similarity and differences with other work. With respect to the efficiency of RE-PHINE, the time required by decompression has already been presented. Experiments in Chapter 5 showed that the total decoding time for NEWS was 236.9 seconds, when SHUFF was used as the coder (Table 5.12 on page 149). For the purpose of RE-PHINE, less time is required since only the phrase hierarchy and the two lexicons are fully decoded. Decoding the two lexicons required 0.8 seconds (Table 5.9 on page 147). The total time required for phrase hierarchy decoding, including graph construction by RE-PHINE was 12.3 seconds, averaged over three trials on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache. Experiments in Chapter 6 showed that searching for symbols in the reduced word sequence encoded with RE-VIEW16 is fast, but depends on the number of
Phrase Browsing

symbols searched for (see Figures 6.3 and 6.4 on pages 182 and 184, respectively). The values in these graphs provide a good estimate of the amount of time required to search a sequence coded with Re-View16. Finally, locating 100 positions in a stream of modifiers with indexed Huffman coding requires no more than 2.2 elapsed seconds if a block size of 32,768 bytes was chosen, shown in Table 6.4 on page 177.

In summary, 13.1 CPU seconds is required by Re-Phine to decode the phrase hierarchy and the two lexicons. Approximately 10 elapsed seconds is necessary for Re-View16 to search the compressed sequence for 100 different symbols. The modifiers at a certain location in a stream can be found in less than 0.1 seconds.

It is more difficult to assess the effectiveness of Re-Phine, or phrase browsing in general. The main feature of phrase browsing is the user-driven navigation between symbols in the phrase hierarchy. This interface experience cannot be easily quantified. If the searching phase of Re-Phine is assessed using precision as the evaluation metric, then problems similar to those encountered with Boolean queries need to be addressed, since every result returned contains the symbol being sought, and precision is 1.0.

On the other hand, the proportion of relevant results (recall) can be measured, but in a slightly different manner than in the traditional definition. Unlike with a Boolean query or an exhaustive string search, recall is not guaranteed to be 1.0, because of the way in which the pairing process of Re-Pair breaks up phrases. For example, suppose part of a message contains the five symbols: αβδβδ. Suppose also that the frequency of αβ throughout the message is higher than the pair βδ, and that the replacements made are ① → αβ followed by ② → βδ. With these replacements, the segment αβδβδ reduces to ①δ②. As a consequence, if the symbol ② is selected during phrase browsing, then only the second occurrence of it in this segment is found. The first occurrence has been split by the replacement with symbol ①, and is not reported to the user.

Experiments were conducted with the first 20 MiB of News (WSJ20), compressed with punctuation-aligned Re-Pair, to evaluate the average recall. Unlike results from Table 4.12 on page 99, phase 1 of Re-Merge has been applied, removing 16 duplicates in the phrase hierarchy. For each symbol in the phrase hierarchy, the frequency of the
7.4 Evaluating Re-Phine

<table>
<thead>
<tr>
<th>Symbol length</th>
<th>Number of symbols in phrase hierarchy $(P)$</th>
<th>Distribution of recall values</th>
<th>Median recall</th>
<th>Mean recall</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39,637</td>
<td>1.000</td>
<td>1.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>79,693</td>
<td>0.750</td>
<td>0.706</td>
<td>0.295</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>60,825</td>
<td>1.000</td>
<td>0.794</td>
<td>0.260</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>25,245</td>
<td>1.000</td>
<td>0.862</td>
<td>0.225</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10,932</td>
<td>1.000</td>
<td>0.917</td>
<td>0.184</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4,998</td>
<td>1.000</td>
<td>0.938</td>
<td>0.163</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2,895</td>
<td>1.000</td>
<td>0.954</td>
<td>0.141</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1,686</td>
<td>1.000</td>
<td>0.954</td>
<td>0.144</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1,180</td>
<td>1.000</td>
<td>0.957</td>
<td>0.136</td>
<td></td>
</tr>
<tr>
<td>10+</td>
<td>4,454</td>
<td>1.000</td>
<td>0.955</td>
<td>0.136</td>
<td></td>
</tr>
<tr>
<td>Overall</td>
<td>231,545</td>
<td>0.975</td>
<td>0.904</td>
<td>0.168</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Recall of the symbols in the \texttt{WSJ20} phrase hierarchy. Median recall values for each symbol length are in the third column, followed by the mean, and the standard deviation.

symbol and all other higher generation symbols that contained it were compared with the actual frequency in the word sequence. The results from these experiments are presented in Table 7.1.

In Table 7.1, the 231,545 distinct symbols formed from \texttt{WSJ20} were separated into ten symbol groupings based on their expanded length, measured in word tokens. Since the zero-length word token for separating a long non-word token is of no interest, it does not appear in the table.

The second column of the table shows the size of each grouping. The majority of symbols were 2 or 3 word tokens in length. The remaining three columns indicate some gross statistics for the distribution of recall values for these symbols: the median, the mean, and the standard deviation. Generally, results were worse for symbols which represented 2 to 4 word tokens. For the remaining categories, the average recall was at least 0.9 and their corresponding medians meant that at least half of the symbols had a recall of 1.000. This result is further reinforced by the standard deviation ranging from 0.000 to 0.184. The average recall and standard deviations were less satisfactory for symbols lengths of 2 to 4. When the symbol length is 2, the median recall was only 0.750.

The results from the table can be interpreted as follows. Short phrases are more
Phrase Browsing

difficult to accurately find than primitives and long phrases. For example, in WSJ20, the phrase “prime minist”, can be easily split depending on the words immediately before and after it. Note, however, that the most common phrase that includes each word is exactly the one most likely to have a high recall. And, as phrases get longer, since their underlying components are more frequent, RE-PAIR does not separate them.

One anomaly is not shown in the experiments with RE-PHINE. Even though precision across the set of symbols in the phrase hierarchy is always 1.0, not every symbol can be searched. In particular, while RE-PHINE is able to locate every symbol chosen through phrase browsing, many combinations of words do not exist in the phrase hierarchy and cannot be searched for. In Figure 7.4(c), the phrase “prime minist” was extended to the left, and one phrase found was “canadian prime minister”. However, the phrase “canadian prime” does not exist in the phrase hierarchy, and therefore, cannot be searched for by RE-PHINE. The problem is the symbol pairing order employed by RE-PAIR (and RE-MERGE) makes no attempt to place every multi-symbol combination into the phrase hierarchy. This point is expanded further later in this chapter.

Phrases for the dictionary are selected by RE-PAIR and RE-MERGE based solely on symbol frequency, and in this sense, RE-PHINE is closely related to SEQUITUR’s phrase browser, PHIND. Nevill-Manning et al. [1997] identified the phrase boundary problem with the phrase hierarchies produced by SEQUITUR; a problem which also affects the phrase hierarchies of RE-PAIR. The phrase boundary problem refers to two symbols which expand to identical strings, but have differing compositions. For example, it is possible that “canadian prime | minist” and “canadian | prime minist” become two separate symbols in the phrase hierarchy, creating no problems for compression, but confusing the browsing process. On the other hand, phase 1 of RE-MERGE explicitly addresses this problem when phrase hierarchies are merged. Recall that phase 1 expands each symbol into a string of primitives for comparison. Symbols that represent duplicate strings are removed.

Another drawback with RE-PHINE is apparent in the first screen of Figure 7.4(a), which presents a list of the word tokens in the document collection. Such an exhaustive list is similar to a Boolean or ranked query system which lists every word found in the collection,
7.4 Evaluating Re-Phine

<table>
<thead>
<tr>
<th>Case fold</th>
<th>Unique primitives</th>
<th>Average length (chars)</th>
<th>Percentage of primitive extensions</th>
<th>Unique phrases</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>533,205</td>
<td>8.00</td>
<td>10.2% 11.3% 18.5% 60.0% 8,464,282</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>448,278</td>
<td>8.10</td>
<td>9.6% 11.4% 18.8% 60.2% 8,269,436</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>401,448</td>
<td>7.34</td>
<td>10.3% 11.7% 18.2% 59.8% 8,147,217</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>336,739</td>
<td>7.46</td>
<td>9.7% 11.8% 18.1% 60.4% 7,940,620</td>
</tr>
</tbody>
</table>

Table 7.2: The effect case folding and stemming has on the phrase hierarchy created from punctuation-aligned Re-Pair for NEWS with method E of Re-Merge. The last row corresponds to the application of both case folding and stemming of word tokens, and reflects the arrangement used for the experiments with NEWS in this chapter.

and is analogous to a book index containing every word found in the book. The advantage of Re-Phine for phrase browsing is not the word lexicon, but the relationships formed between words in order to combine into longer phrases. Nevertheless, such an opening window is distracting to users. Subsequent lists also suffer from the same problem, but to a lesser degree. The words in the first window and phrases in the remaining windows of Figure 7.4 could be filtered so that unhelpful symbols are removed.

Case folding and stemming by Re-Pair reduce the size of the word lexicon and phrase hierarchy, while also improving browsing by providing one word which represents several similar ones. Chapter 4 showed the effect stemming and case folding had on the the word lexicon of WSJ20 (Table 4.4 on page 87). Table 7.2 updates those results for NEWS. Recall that the rules for stemming only apply to words in lower case. In the third row of the table, if the part of a word in upper case is examined by the stemming algorithm, and no case folding is applied first, then the word would be left unchanged. For example, “Searching” stems to “Search”, while “SearchInG” does not. The third column of the table shows the number of primitives in the phrase hierarchy, while the last column indicates the number of phrases. Case folding and stemming has an effect on the size of the word lexicon, confirming the results for the smaller WSJ20 file. Moreover, the average lengths of words in the lexicon are slightly longer than the values listed in Table 4.4, as indicated in the fourth column of the table.

In the fifth, sixth, seventh, and eighth columns of the table, the set of primitives has been divided into four groups based on whether or not each one can be extended. In all
cases, about 60% of the symbols do not participate as a component of a higher generation symbol. However, simply eliminating these symbols from the opening window of Re-Phine is too extreme. If a word token fails to become part of a phrase, it does not mean that the word is not interesting or is infrequent.

Nevill-Manning et al. [1997] noticed a similar problem with the phrase browser Phind. The solution adopted was to filter words and phrases based on word categories. Words were labelled as rare, interesting, or common based on a set of frequency thresholds. When the phrase hierarchy is explored, an expanded phrase is shown only if it differs from the previous phrase by at least one interesting word. For example, if “book” is an interesting word, but “the” is a common word, then the expansion of “book” to “the book” would not be displayed. The reason is that adding a common word to the original word does not add any value for the user. However, “book” can be expanded directly to “the new book”, provided the word “new” is also an interesting word. The drawback of this heuristic is that a common word may still be of interest to users, and a rare word does not necessarily represent a typographical error.

At the centre of the problem with Sequitur, and Re-Pair as well, is that selecting phrases based on frequency generates too many phrases, with many differing by only a common word like “the”. Paynter et al. [2000] proposed the solution of combining Sequitur with Kea, since Kea was known to select too few phrases by itself. Likewise, the combination of Re-Pair with a mechanism similar to Kea may improve the number of phrases presented by Re-Phine. A more detailed look at the phrases created by Re-Pair for Re-Phine is provided next.

Comparing Re-Phine with Noun Phrases

Another part of Re-Phine that is worth evaluating is the usefulness of the phrases presented to the user. The quality of the phrases is not related to precision or recall because the primary concern is the type of phrases available for browsing, and not the result from searching.
7.4 Evaluating Re-Phine

Wolff [1980] produced phrases through a recursive pairing mechanism called MK10. In MK10, pairs are chosen based on the part-of-speech categories assigned to the word tokens. The quality of the phrases was compared to the phrases selected by a human linguist. In a similar fashion, the phrases made available by RE-PAIR can be compared with the ones identified by a reference program employing natural language processing (NLP) techniques.

As with other IR systems based on NLP techniques, the basis for these experiments is that a user is more interested in a complete noun phrase such as “prime minister” than a broken phrase like “Canadian prime”. The aim of the experiments is to report on the probability that a symbol in the phrase hierarchy is a noun phrase, and the probability that noun phrases exist in the phrase hierarchy. Noun phrases are identified through an NLP technique called \textsc{link grammar} [Sleator and Temperley, 1991] embodied in a program by the same name, available from http://www.link.cs.cmu.edu/link/. The latest version of the software is 4.1, dated August 2000.

The \textsc{link grammar} system processes a sentence at a time. In the first step, individual words are assigned to part-of-speech categories by consulting a static dictionary. The categories of novel words are guessed through a technique the authors call “morpho-guessing”. Words have links to both sides for combining with other words in the sentence. For example, the two words “the book” can merge because of a right-facing “D+” link on the word “the” and a corresponding left-facing “D−” link on the noun “book”. The “D” in both cases symbolise a determiner such as “the” and “an”. An edge is drawn between these two words to indicate that they are compatible with each other. When a grammatical sentence is correctly parsed, a valid linkage is said to be formed. Several requirements must be satisfied in order to obtain a linkage. One of these requirements is that the graph formed must be connected, with all of the words in the sentence reachable.

Subsequent work on the system [Grinberg et al., 1995] permits \textit{null links} and post-processing of valid linkages with a phrase parser for identifying constituents such as noun phrases and verb phrases. If a sentence cannot be successfully parsed, then words in the sentence are removed incrementally until a successful parsing can be found. When the
Phrase Browsing

sentence is finally parsed, each removed word introduces a null link into the linkage. When a linkage for a sentence has been established, the post-processing phrase parser combines words to form constituents of one or more words. On-line documentation for the phrase parser\(^2\) reports that the constituents determined by the parser were 75% correct on the Penn Treebank text\(^3\), which is newspaper text already annotated linguistically.

A sample parsing of a headline from WSJ20 is shown in Figure 7.7. In the figure the headline is shown in part (a). The LINK GRAMMAR system considers it a valid sentence since two linkages are identified, as shown in parts (b) and (c). In Figure 7.7(a), the “G” link is used to pair “South” with “Korea”. A “G” link connects proper nouns together, identified by the upper case letters at the beginning of each of these words. Similarly, a “G” link is inserted between “Current” and “Account”. The second linkage case folds “South” so that the first “G” link is converted to an “A” link. An “A” link connects an adjective with the noun it follows. The “Ss” link pairs a subject with a verb, while “Ost” edges connect a transitive verb to a singular object. So, LINK GRAMMAR incorrectly treats “s” as a verb in both linkages. Linkages are scored so that ones with less null links or a lower edge length total are displayed first. The constituents for the linkage in (b) is presented in Figure 7.7(d). At the beginning of each constituent is its classification. The abbreviation “VP” denotes a verb phrase, while “NP” is a noun phrase.

As with previous work in IR and NLP, the focus of the phrase comparison is with noun phrases. The number of unique phrases that LINK GRAMMAR found in WSJ20 was compared with RE-PHINE’s phrase hierarchy. Since only grammatical sentences can produce linkages, the test data of news articles were filtered so that sequences of words that are not expected to give many noun phrases were omitted. First, SGML tags were removed and multiple consecutive whitespaces collapsed into a single space. Sentences were given to LINK GRAMMAR one at a time. Then, the constituent analysis was determined by the phrase parser.

As with the earlier experiments to measure recall, the WSJ20 document was used as

\(^2\)Available at http://www.link.cs.cmu.edu/link/ph-explanation.html.

\(^3\)The Penn Treebank project is at http://www.cis.upenn.edu/~treebank/.
Section 7.4 Evaluating Re-Phine

![Diagram of South Korea's Current Account](image)

Figure 7.7: An example parsing of a headline from WSJ20 using the LINK GRAMMAR system. The headline is shown along the top as part (a). Two linkages are determined by LINK GRAMMAR, which are shown in (b) and (c). The linkage from (b) is post-processed for constituent identification, yielding the output in (d). Noun phrases are marked with "NP".

test data. The computing resources of the LINK GRAMMAR program were quite high. It required just over 5 hours to process the first 1 MiB of WSJ20 on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache test machine. This time included the above steps, starting from the SGML tag filtering, and ending with the identification of constituents. Note that the amount of work performed by LINK GRAMMAR is more than what is necessary to identify noun phrases. Other systems, such as the KEYPHIND browser of Gutwin et al. [1999], simply categorise words according to their parts-of-speech, and then form noun phrases based on a set of fixed rules. Even so, Gutwin et al. required 4 days to process 1 GiB of text on a Pentium 233 MHz machine running Linux. In contrast, LINK GRAMMAR must first successfully find a linkage for a sentence, and only when this is done, can it isolate the noun phrases.
Phrase Browsing

<DOC>
<DOCNO> WSJ870323-0181 </DOCNO>
<HL> South Korea's Current Account </HL>
<DD> 03/23/87 </DD>
<SO> WALL STREET JOURNAL (J) </SO>
<IN> FREST MONETARY NEWS, FOREIGN EXCHANGE, TRADE (MQN) </IN>
<DATELINE> SEOUL, South Korea </DATELINE>
<TEXT>
South Korea posted a surplus on its current account
of $419 million in February, in contrast to a deficit
of $112 million a year earlier, the government
said. The current account comprises trade in goods
and services and some unilateral transfers.
</TEXT>
</DOC>

Figure 7.8: An example article from the WSJ20 document, with noun phrases indicated with outlined boxes. Shaded boxes represent parts of the article that were removed prior to applying LINK GRAMMAR. The article has been formatted with linefeeds for improved display.

After LINK GRAMMAR completes, constituents which are noun phrases, and are not composed of other phrases, were recorded. The noun phrases were case folded and then stemmed with the Porter stemming algorithm employed by PRE-PAIR. In order to comply with the 16 character per word limitation imposed by PRE-PAIR, spaces were inserted into long words. Finally, a list of all of the distinct noun phrases in WSJ20 was prepared.

Figure 7.8 illustrates how a news article in WSJ20 was parsed, with linefeeds altered to improve display. Text that was stripped from the article is shown shaded. All SGML tags are removed, as well as text within most tags. Only text between opening and closing <HL> and <TEXT> tags were retained, since the headline and the body of the article has the best chance of providing complete sentences. The noun phrases found by LINK GRAMMAR for this example are shown in boxes. The headline used as an example in Figure 7.7 was taken from the article shown in Figure 7.8.

The LINK GRAMMAR system successfully found 147,294 grammatical sentences in WSJ20. Within these sentences, 744,721 noun phrases were identified, of which 216,661 were unique after stemming and case folding. These unique phrases were divided into

214
7.4 Evaluating Re-Phine

<table>
<thead>
<tr>
<th>Phrase length</th>
<th>Number of phrases in hierarchy (P)</th>
<th>Number of unique noun phrases (NP)</th>
<th>Size of P \cap NP</th>
<th>Probability P has noun phrase in P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39,637</td>
<td>13,172</td>
<td>13,172</td>
<td>0.332</td>
</tr>
<tr>
<td>2</td>
<td>79,693</td>
<td>78,527</td>
<td>21,692</td>
<td>0.272</td>
</tr>
<tr>
<td>3</td>
<td>60,825</td>
<td>72,492</td>
<td>6,350</td>
<td>0.104</td>
</tr>
<tr>
<td>4</td>
<td>25,245</td>
<td>32,419</td>
<td>1,578</td>
<td>0.063</td>
</tr>
<tr>
<td>5</td>
<td>10,932</td>
<td>12,304</td>
<td>397</td>
<td>0.036</td>
</tr>
<tr>
<td>6</td>
<td>4,998</td>
<td>4,939</td>
<td>144</td>
<td>0.029</td>
</tr>
<tr>
<td>7</td>
<td>2,895</td>
<td>1,696</td>
<td>32</td>
<td>0.011</td>
</tr>
<tr>
<td>8</td>
<td>1,686</td>
<td>662</td>
<td>8</td>
<td>0.005</td>
</tr>
<tr>
<td>9</td>
<td>1,180</td>
<td>247</td>
<td>2</td>
<td>0.002</td>
</tr>
<tr>
<td>10+</td>
<td>4,454</td>
<td>203</td>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>Total</td>
<td>231,545</td>
<td>216,661</td>
<td>43,376</td>
<td>0.187</td>
</tr>
</tbody>
</table>

Table 7.3: Quantifying the quality of symbols in the phrase hierarchy of W3J20 by calculating the intersection of noun phrases, as identified by LINK GRAMMAR, with the phrase hierarchy. Symbols have been divided into 10 groups, based on length (in words). The second and third columns provide insight into the number of symbols (or noun phrases). The intersection of these two columns are reported in the fourth column. The last two columns indicate the proportion of phrases in P which are noun phrases (fourth column divided by the second column), and the probability a noun phrase also being in P (fourth column divided by the third column), respectively.

groups based on their lengths, in words, and compared with the phrases available to Re-Phine, which were also grouped according to length. The phrases for Re-Phine are identical to the ones reported in Table 7.1. Information about these two sets of phrases are shown in Table 7.3. The sizes of the sets of phrases are listed by lengths in the second and third columns. Note that noun phrases are only found if a valid linkage has been determined. In the fourth column of this table, the number of phrases common to the phrase hierarchy and in the set of noun phrases has been tabulated. The probability of a symbol in P being a noun phrase is given in fifth column. The reverse, the probability of randomly selecting a noun phrase which is in the phrase hierarchy, is listed in the last column of the table.

As Table 7.3 shows, the majority of phrases in the phrase hierarchy and noun phrases from LINK GRAMMAR are 2 or 3 words in length. The overlap between the two sets is around 28% for phrases of length two for both measures, a somewhat disappointing result. Worse, as the phrases increase in length, there is even less correlation with the phrase
Phrase Browsing

hierarchy, as the last two columns show. This difference is partly due to the methods used by Re-Pair and Link Grammar to build phrases. The Link Grammar system operated on sentences individually, so phrases were guaranteed to be aligned on sentence boundaries. Also, SGML markup did not appear in any of the sentences, because of the preliminary filtering. On the other hand, while punctuation-aligned Re-Pair encourages alignment with punctuation, phrases can still span sentences, since they are not processed individually. Furthermore, since Re-Pair was applied on the original version of WSJ20, any phrase that contained markup would not have a corresponding phrase from Link Grammar. The analysis in Table 7.3 is based solely on the existence of phrases, and takes no account of the frequency. Because Re-Pair finds common phrases, it is likely that the scores would be considerably better if the probabilities were assessed over the sequence instead of over the phrase hierarchy.

Making phrases available in the phrase hierarchy is only half of Re-Phine’s purpose since matching contexts must also be located in the reduced sequence. Using the same methodology as established for Table 7.1, the recall of the phrases that exist in both the phrase hierarchy and the list of noun phrases is presented in Table 7.4. This time, rather than presenting recall values for all symbols in the phrase hierarchy, only the ones that were also noun phrases identified by Link Grammar are considered. The second column of this table lists the number of noun phrases that were also in the phrase hierarchy, copied from the fourth column of Table 7.3. In Table 7.4, as with the earlier experiment, the median recall value for each phrase length group is reported. The mean and standard deviation of each set of recall values is also shown.

The results from Table 7.4 mirror those of Table 7.1, but note the small size of the sample for phrases of length 7 or more. Again, the average recall is above 0.7, and more than half of the symbols had perfect recall. The standard deviation is wider and affects more groupings than in Table 7.1.
7.5 Tools for Retrieval

<table>
<thead>
<tr>
<th>Phrase length</th>
<th>$\mathcal{P} \cap \mathcal{NP}$</th>
<th>Distribution of recall values</th>
<th>Median recall</th>
<th>Mean recall</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13,172</td>
<td></td>
<td>1.00</td>
<td>1.00</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>21,692</td>
<td></td>
<td>0.80</td>
<td>0.720</td>
<td>0.298</td>
</tr>
<tr>
<td>3</td>
<td>6,350</td>
<td></td>
<td>1.00</td>
<td>0.787</td>
<td>0.271</td>
</tr>
<tr>
<td>4</td>
<td>1,578</td>
<td></td>
<td>1.00</td>
<td>0.845</td>
<td>0.242</td>
</tr>
<tr>
<td>5</td>
<td>397</td>
<td></td>
<td>1.00</td>
<td>0.831</td>
<td>0.264</td>
</tr>
<tr>
<td>6</td>
<td>144</td>
<td></td>
<td>1.00</td>
<td>0.849</td>
<td>0.250</td>
</tr>
<tr>
<td>7</td>
<td>32</td>
<td></td>
<td>1.00</td>
<td>0.853</td>
<td>0.258</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td>1.00</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td></td>
<td>1.00</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>10+</td>
<td>1</td>
<td></td>
<td>1.00</td>
<td>1.000</td>
<td>0.000</td>
</tr>
<tr>
<td>Overall</td>
<td>43,376</td>
<td></td>
<td>0.980</td>
<td>0.889</td>
<td>0.158</td>
</tr>
</tbody>
</table>

Table 7.4: Recall effectiveness of the noun phrases in the phrase hierarchy for WSJ20. The median has been calculated based on phrase length (in words) in the third column, with the mean, and the standard deviation indicated in the last two columns.

7.5 Tools for Retrieval

The Re-Phine system allows users to perform two inter-related tasks. The first of these is phrase browsing. Starting from a list of the distinct words found in the document, the user can either extend the current symbol by adding symbols to the left or to the right, or decompose the current symbol into its two constituents. A graph structure with one symbol from the phrase hierarchy at each node was established for the purpose of phrase browsing. Each node has six pointers for navigation and defining its structure.

Once the user has found a phrase of interest, the graph structure is consulted again to determine the complete set of phrases containing the chosen one. This set of symbols, $C$, is the basis for the search in the reduced word sequence to determine all of the locations at which any symbol in $C$ appears, along with a suitable context for each. The second job of Re-Phine is to interactively display this set of results one by one to the user. Each result consists of word tokens which have been stemmed and case folded. The non-words that follow each word token are also initially unavailable. Each presented result can be refined by the user by requesting modifiers from the three modifier streams, in any combination. If at this stage, another symbol of interest has been isolated, it can be passed back to the
phrase browsing part of RE-PHONE.

In this chapter, RE-PHONE has been demonstrated with a document collection 1,000 MiB in size on a 933 MHz Pentium III with 1 GiB RAM and 256 KiB on-die cache. Before considering larger collections, note that only three of the seven streams available to RE-PHONE must reside in memory. The four remaining streams are compressed in blocks so that only parts of them are read from disk into memory. The phrase hierarchy and the two lexicons occupy 1.8% of disk space compared to the size of NEWS, and all of them must be accessible by RE-PHONE in decoded form. The complexity and size of the phrase hierarchy ensures that it requires the most memory during browsing after it has been fully decoded.

Based on these observations, several approaches may be selected in order to browse larger collections. First, Section 5.7 showed how RE-MERGE can be extended so that documents are appended to a compressed representation while keeping the phrase hierarchy static. Experiments demonstrated this technique with character-based RE-PAIR, and an analysis of the changes necessary for punctuation-aligned RE-PAIR was discussed. In particular, a method of handling novel words is required. Second, either the maximum generation or the maximum symbol length can be limited so that the phrase hierarchy only contains moderately long phrases. Third, while all word tokens must exist in the phrase hierarchy as primitives, additional heuristics, such as those offered by KEA and LINK GRAMMAR, may be used when building the phrase hierarchy. Instead of reducing the number of phrases shown by RE-PHONE, this approach would prevent certain symbols from being added to the phrase hierarchy. Finally, any combination of these three methods can be used in order for larger document collections to be compressed by RE-PAIR and RE-MERGE, and subsequently browsed.

The RE-PHONE implementation has not reached the stage where it could sensibly be made the subject of a user study. As with SEQUITUR, too many symbols exist in RE-PAIR’s phrase hierarchy, and some filtering is necessary to improve usability. The removal of symbols through NLP techniques has been trialled by other IR systems, and it is likely that RE-PHONE would be more feasible if similar techniques could be employed. An
extension of this idea is to rank symbols using some metric so that those that are expected to be the most useful are presented at the top. Since there are reasons why users would want frequent and and infrequent phrases, simply sorting symbols by frequency would not help.

User studies have been conducted to quantify the usefulness of phrase browsing retrieval systems [Gutwin et al., 1999, Jones and Paynter, 2001], with some results being in favour of phrase browsing, and some less conclusive. Wacholder and Nevill-Manning [2001] concluded that more work is still required in the area of phrase browsing evaluation with users. But even if the suggested changes were adopted for RE-PHINE, it is expected that a comparative user study would not favour RE-PHINE. While providing users with a phrase browser was one of the aims of this development, the necessary tension between browsing and compression inevitably means that RE-PHINE is not as careful as a dedicated phrase searching mechanism. One necessary improvement appears to be achieving higher recall. It is possible that the solution lies in closely regulating word-based RE-PAIR’s phrase selection heuristic. As word-aligned RE-PAIR showed, preventing certain pairs of symbols from joining does not necessarily degrade compression effectiveness (see Figure 4.5 on page 103).

The benefits to compression are easier to quantify than the benefits of phrase browsing. Since the latter quantity cannot be measured, it is also difficult to assess when a reasonable compromise between the two has been achieved. Instead, the goal of this thesis has been to identify areas where compression and retrieval effectiveness and efficiency are lost, so that one can be adjusted in favour of another, as necessary.

RE-PHINE is built on top of a compression mechanism, a structure which limits its ability to browse and retrieve data. Throughout this thesis, the amount of compression effectiveness lost in favour of faster retrieval times has been highlighted as a means of compromising between the two competing sets of constraints. In this chapter, the current drawbacks with RE-PHINE represent the amount of retrieval functionality lost in favour of supporting the lossless compression of a document collection with the underlying mechanism. Employing NLP techniques with RE-PHINE may be the ideal solution for
Phrase Browsing

compensating for RE-PHINE’s loss in retrieval effectiveness. As mentioned above, at the same time, NLP techniques may also reduce the number of symbols that reside in memory during browsing.

Regardless of the improvements made to RE-PHINE, phrase browsing in general is one of many IR techniques. This chapter has also provided a glimpse into other querying techniques, including ranked queries, proximity queries, and Boolean queries. The system a user chooses depends on factors such as how familiar the user is with the document, whether query terms include phrases, and whether the order and the positions of the words in a phrase matter. As Gutwin et al. [1999] pointed out, no single IR method is perfect and the most viable solution might be to provide “several search tools that can make up for each others’ shortfalls”.

This chapter concludes the discussion in this thesis about the balance between compression and phrase browsing. Before summarising the thesis in Chapter 9, the next chapter discusses a separate topic related to the compression of small HTML files.
Chapter 8

Compression of HTML Documents

This chapter deals with the compression of documents published on the World Wide Web using the HyperText Markup Language (HTML) [Raggett et al., 1999]. While the HTML standard is a subset of the SGML standard, HTML documents have many similarities with the SGML documents considered for most of this thesis. The NEWS document is 1,000 MiB in size, but artificially constructed by concatenating news articles together. When the compression systems outlined in previous chapters were applied to NEWS, the best compression ratio achieved was 1.674 bpc for PPMD (Table 2.5 on page 41).

Typical HTML files cannot be simply concatenated together and compressed if they are to be accessed on an individual basis. While HTML files are as brief as short news articles, news articles do not form relationships with other articles, except for their chronological order. In addition, “flattening” the HTML files on a web server into a single document would lose the intricate hierarchical structure created by hyperlinks. On the other hand, most compression systems attain better compression effectiveness as more text is available as context.

In the case of the smaller HTML files, context exists through hyperlinks. This chapter describes an investigation into a scheme under which HTML files from a medium-sized web server are compressed for storage and transport by exploiting the way they are accessed, either with or without hyperlinks.
Compression of HTML Documents

The rest of this chapter is laid out as follows. In Section 8.1, related work by other researchers is summarised. Then, in Section 8.2, an overview of the data collected from the web server is given. In the context of this data, the experimental framework is described in Section 8.3, followed by the results in Section 8.4. Some conclusions about this work is drawn in Section 8.5.

8.1 Related Work

The two areas covered by this chapter are the reduction of bandwidth through compression and the detection of context in web pages. Both of these areas have been studied extensively by others.

Several techniques have been devised which reduce the bandwidth requirements of HTML documents. Delta encoding (for example, see Mogul et al. [1997, 1998]) assumes that users usually request a recent version of a previously downloaded file. Instead of re-sending the entire file, a difference between the latest file and the version of the file in the client’s cache is sent. Nielsen et al. [1997] considered the benefits of lossy compression of HTML files by case folding tags without affecting the web browser’s ability to render the document. Nielsen et al, also demonstrated that high-level compression of documents with the zlib library [Gailly and Adler, 1995] was more effective than the compression systems built into the set of modems that they considered.

On the World Wide Web, the current version of the HyperText Transfer Protocol (HTTP/1.1) [Gettys et al., 1999] supports compression through two header fields. The Accept-Encoding field allows the browser to notify the web server of its support for one of a set of encodings, as defined by the HTTP/1.1 standard. These encodings are primarily used for compression, and include both gzip and deflate. If the web server is able to support the specified encoding, then a compressed version of the requested document is sent to the client, and the Content-Encoding header field is embedded within the response.
8.2 Data Overview

Support for these headers have been implemented by the `mod_gzip` module\(^1\) for the Apache web server\(^2\), used by approximately 63% of the web sites on the Internet, according to a September 2003 survey conducted by Netcraft\(^3\). The current version of `mod_gzip` (1.3.26.1a) delivers compressed documents from a cache, but can recompress files during a request when newer versions are found.

With respect to hyperlinks, Brin and Page [1998] applied hyperlinks to IR by using them to judge the relevance of documents on the Web. Mizuuchi and Tajima [1999] established a set of rules to determine the most likely path a user would take to reach a certain HTML file. Li et al. [2000] later applied a set of rules to a web site in order to divide it into a set of logical domains, with each domain relevant to the same topic.

The experiments in this chapter are concerned with the compression of HTML files based on the order in which they are accessed by users.

8.2 Data Overview

Rather than developing a compression mechanism embedded within multiple web browsers and a web server, a simulation was performed. The simulation made use of access logs and web pages from the medium-sized web server of the University of Melbourne’s Department of Computer Science and Software Engineering\(^4\). The web server used version 1.1 of HTTP.

Access logs lasting one week were taken, starting from 12.15 a.m. on 17 July 2000 until 12.15 a.m. on 23 July 2000. The access logs are in the Apache Common Log Format\(^5\). Each entry in the log records the client that accessed the server, the file requested, and the date and time of the request. While version 1.1 of HTTP supports persistent connections, the log file does not indicate this. Because of this, it is not possible to determine from

\(^1\)http://sourceforge.net/projects/mod-gzip/
\(^2\)http://httpd.apache.org/
\(^3\)http://news.netcraft.com/
\(^4\)http://www.cs.mu.oz.au/
\(^5\)Described at http://httpd.apache.org/docs/logs.html
Compression of HTML Documents

<table>
<thead>
<tr>
<th>File type</th>
<th>Number of requests</th>
<th>Total volume (MiB)</th>
<th>Average (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local requests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTML text</td>
<td>55,976</td>
<td>626.4</td>
<td>11.5</td>
</tr>
<tr>
<td>images</td>
<td>20,884</td>
<td>73.6</td>
<td>3.6</td>
</tr>
<tr>
<td>other</td>
<td>1,871</td>
<td>76.4</td>
<td>41.8</td>
</tr>
<tr>
<td>External requests</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTML text</td>
<td>48,343</td>
<td>443.1</td>
<td>9.4</td>
</tr>
<tr>
<td>images</td>
<td>83,585</td>
<td>304.5</td>
<td>3.7</td>
</tr>
<tr>
<td>other</td>
<td>5,739</td>
<td>637.4</td>
<td>113.7</td>
</tr>
<tr>
<td>Total</td>
<td>216,378</td>
<td>2,161.5</td>
<td>10.2</td>
</tr>
</tbody>
</table>

Table 8.1: Breakdown of requests by origin and file type, 17 July 2000 to 23 July 2000 inclusive.

<table>
<thead>
<tr>
<th>File type</th>
<th>Number of files</th>
<th>Total size (MiB)</th>
<th>Average (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML text</td>
<td>68,361</td>
<td>721.0</td>
<td>10.8</td>
</tr>
<tr>
<td>images</td>
<td>16,840</td>
<td>300.0</td>
<td>18.2</td>
</tr>
<tr>
<td>other</td>
<td>17,513</td>
<td>2,233.8</td>
<td>130.6</td>
</tr>
<tr>
<td>Total</td>
<td>102,714</td>
<td>3,254.8</td>
<td>32.4</td>
</tr>
</tbody>
</table>

Table 8.2: Snapshot of a web site, 24 July 2000.

the log file whether two consecutive requests from the same client was made by the same person, or by two different people using the same computer.

To prepare these logs for our investigation, the entries in the access logs were separated based on whether the client was local or external to the department, and then, whether the document requested was an HTML file, an image, or another file type. The results from this breakdown are summarised in Table 8.1.

A snapshot of the web site was taken on 24 July 2000, soon after the access logs were acquired. The snapshot was also split based on file type, yielding Table 8.2. For each file type, the number of files, the total file size, and the average file size are indicated.

The data described in these two tables forms the basis of the simulation. Almost 40% of the total requests were made from within the same network as the web server. Since transmission costs of local requests are relatively less than external requests, these
local requests have been omitted from the simulation. Furthermore, as the topic of this investigation is HTML files, the 443.1 MiB of HTML files requested from an external client is the focus of the work. The remaining externally requested files are used in the simulation, but are of lesser importance.

The size of the HTML files requested reflects their average size on the web server. In both cases, the average HTML file size was around 10 KiB. According to Table 8.2, most of the files on the web server were HTML files, but they occupied less than one quarter of the total disk space. The files labelled “other” include software packages (often in large archives) and documents created in proprietary file formats.

### 8.3 Experimental Framework

The purpose of the simulation was to establish the extend to which compression effectiveness could be improved, using the logs of requested files as an indication of typical access sequences. Suppose $x^i$, $y^i$, and $z^i$ are HTML files at time $t$. If a user obtains $x^i$ and then $y^j$, where $i < j$, then the straightforward approach to achieving bandwidth reduction is to compress each file individually. However, after $x^i$ has been transmitted, the web server can improve the compression of $y^j$ by making use of $x^i$, a file that is now assumed to exist on both sides of the communications channel. The compressed representation of $y^j$ given $x^i$ (denoted as, $c(y^j|x^i)$) is transmitted instead of $c(y^j)$ alone. The compression system is said to be “primed” with $x^i$ before $y^j$ is processed.

Because relatively little structure exists in the access logs, heuristic approximations have to be made with regard to whether two requests from the same client were made by the same user\(^6\). Also, a limit on the difference between the times $i$ and $j$ was imposed in order to reduce the chance that the first file, $x^i$, had not changed. A time interval, $\tau$, was introduced into the simulation, which is the most time a user can take to read a document and move on to the next one. A traversal occurs when a client visits $x^i$ and then $y^j$, provided the difference between $i$ and $j$ is constrained to be less than $\tau$. In the sequence

\(^6\)On a computer where all users share the same cache, this distinction between clients and users is less relevant, though.
of web page accesses by a client, if the interval between any two consecutive requests is less than \( \tau \), then a browsing session for that client has been found.

There are two subtle points to note with respect to these definitions. A valid traversal does not require a hyperlink to be followed, even though the motivation for this work was the hyperlinks within HTML files. This simplification is required because the access logs only indicate the files that were accessed, and not whether they were accessed with hyperlinks, bookmarks, or by manually entering them into a browser. Furthermore, the files within a browsing session can be of any type, but the experiments later are concerned with compression of HTML files within the session. So, the compression of \( y^j \) can be primed with \( x^i \), even though many non-HTML files may be sent between times \( i \) and \( j \). The only restriction is that no more than \( \tau \) minutes has passed between any two adjacent file requests between \( x^i \) and \( y^j \).

In the experiments of this chapter, \( \tau \) was fixed at 5 minutes. The access logs were separated into browsing sessions based on this value of \( \tau \), and in Figure 8.1, only HTML files are reported. The graph indicates the percentage of HTML file requests for various browsing session lengths, measured in number of HTML documents. There were 21,067 browsing sessions found, and with 48,343 HTML files requested overall, with each session containing 2 HTML files, on average. Approximately 30% of the requests were to one-off documents with no traversal prior or following. The remaining 70% of the requests were part of browsing sessions with other HTML files. In one browsing session, there were 2,163 HTML files requested, presumably as part of a search engine crawl. As for the suitability in choosing 5 minutes for \( \tau \), if the threshold was extended to 60 minutes, then the percentage of page requests that were part of a browsing session with only one HTML file falls from 30% to 20%. Given this result, it would seem that 5 minutes is a plausible value.

Instead of using the most recently sent HTML file for priming, the web server and the web browser can also share a file made exclusively for priming. Of course, this priming file must be sent by the web server to every new client which accesses it, and the cost of its transmission must be accounted for in any analysis.
8.3 Experimental Framework

Figure 8.1: Percentage of page requests as a function of browsing session length, in number of HTML files, for \( \tau = 5 \) minutes.

Of the compression mechanisms discussed in this thesis, GZIP was selected as the basis for these experiments for two reasons. First, earlier results in Chapters 2 and 3 have shown that GZIP is relatively fast compared to other systems surveyed – an important factor when the user is waiting in front of a web browser. More importantly, the sliding window mechanism employed by GZIP makes priming possible with little modification, in comparison to other systems. The GZIP system simply compresses the document after the window of phrases has been initialised with the priming text. This method of priming has also been suggested by Witten et al. [1999, pg. 391] for algorithms related to LZ77. Also, priming was investigated with more complex compression systems by Teahan [1998].

There is a one important difference between the priming text for GZIP and the training data employed by RE-MERGE when appending text (see Section 5.7 from page 150). The phrase hierarchy is not altered by RE-MERGE while text is being added. In contrast, the phrases produced by GZIP from the priming text are updated as compression progresses through the file.

Two dictionary files were constructed for the sole purpose of priming GZIP. These two files contain the most frequently occurring words from all of the HTML files on the web server, in decreasing frequency order. After being compressed with the \(-9\) option of GZIP, these two files were 1 KiB and 10 KiB in size. Uncompressed, these two files were 1,583
Compression of HTML Documents

```html
HREF</HTML></A></TITLE></TITLE><P><HTML>http</BODY>
wwwThis<B>RI</B></H1></UL><UL><LI><EM><EM><EM>generate
d<HEAD></HEAD>that2000footerEST<STRONG></STRONG>NA
MESSAGEject</SMALL></SMALL>sortedDateemailhypermaiDT
DtrailerDOCTYPEPUBLICFromthisnextthreadisentisor
```

Figure 8.2: The first 250 bytes of the priming dictionary file, which compresses to 10 KiB. Line-breaks have been introduced to allow display.

bytes and 17,736 bytes in size, respectively. Figure 8.2 shows the first 250 bytes of the larger file.

8.4 Experimental Results

The experiments demonstrate the reduction in the number of bytes transferred by priming the compression mechanism. However, compressing each file before delivery to the client adds a time overhead to file transmission. Each file must be compressed by the web server, transferred, and then decompressed by the client. The impact of this can be minimised if files are compressed while the web server is idle, and before they are requested. Alternatively, compressed files can be placed in a cache of recently requested files. The second of these methods was chosen for the simulation.

The simulation measures the bandwidth and the disk space required at the server. Bandwidth is quantified as the number of bytes sent to the client, while the disk space is the extra overhead required for the cache. It is assumed that all of the web pages exist uncompressed on disk at the server. As pages are requested, they are compressed, sent to the client, and copies of the compressed files are left in the cache, in the hope they can be used again. For the purpose of the simulation, the cache is initially empty and never cleared, so that the total disk space taken is the size of the cache at the end of the simulation.

Nine methods of processing HTML files were evaluated. Compression was executed with gzip, using its -9 option. Two methods, NO-COMPRESSION and INDEPENDENT, form the baseline for the experiments. The NO-COMPRESSION method assumes documents are
sent as plain text, while the INDEPENDENT method compresses each requested HTML document individually, as a stand-alone file.

Then, several other methods were considered. The COMMON-DICTIONARY method compresses each file with either of the two standard dictionary files as priming text (1 KiB or 10 KiB), assuming that the dictionary files are available to the client for "free" and do not need to be explicitly sent. For this reason, the COMMON-DICTIONARY method is unrealistic and the TRANSMITTED-DICTIONARY corrects that by adding the cost of sending the respective dictionary files (in compressed form) for every browsing session. The ONE-PRIOR and the ALL-PRIOR methods utilise the browsing patterns of the users. The ONE-PRIOR method compresses each file, priming GZIP in each case using the previously requested file in the browsing session. The first file in any browsing session is compressed without any priming. The ALL-PRIOR method uses all HTML files in the browsing session for priming. Finally, the ALL-AVAILABLE method is the most optimistic method, since it combines the COMMON-DICTIONARY method with the ALL-PRIOR method. The ALL-AVAILABLE method assumes that a standard dictionary file is available for free and that it can be combined with all previously requested HTML files in that browsing session to prime the compressor.

The results from the simulation are presented in Table 8.3 as percentages. The amount of disk space occupied by the HTML files with no compression is 100%, since it is assumed an uncompressed version of the web pages must exist. The bandwidth taken when no compression is applied is 100%. With each of the remaining compression methods, the disk space usage increases and the bandwidth decreases.

The INDEPENDENT method shows that applying GZIP to files individually already reduces the bandwidth requirements to 27.9%. The disk space rises only by 4.7%, because only a small proportion of the publicly available HTML files were accessed in the simulation window of 7 days. Applying either of the dictionaries to prime GZIP slightly reduces both the disk space and bandwidth requirements in comparison to the INDEPENDENT method. Of course no change in disk space exists between the COMMON-DICTIONARY and the TRANSMITTED-DICTIONARY methods. However, when the more realistic situation of
transmitting the dictionaries is considered, the bandwidth requirements rises from 27.1% to 31.5% for the 1 KiB dictionary, and from 27.2% to 71.2% for the 10 KiB dictionary. The overall cost of transmitting the dictionaries is not recovered - a consequence of the large number of short browsing sessions, and the effectiveness of gzip when dealing with moderate-length files.

The one-prior method is more effective than the previous methods, yet the gain is still marginal. Unlike transmitted-dictionary, no explicit dictionary needs to be sent to the client, and for browsing sessions with only one HTML file, the file is compressed without a priming file. Since the remaining two methods in Table 8.3 require multiple files for priming, maintaining a cache appears difficult due to the many file combinations that could exist. For this reason, disk space consumption is not reported. The bandwidth required by these two methods continues to decrease in comparison with one-prior because on average, the 32 KiB sliding window of gzip can hold more than one HTML file or one dictionary file for priming. Only the experiment with the 10 KiB standard dictionary is shown for the all-available method. It achieves the best result with respect to bandwidth, but this is partly because the dictionary is assumed to be available without charge.

Some of the bandwidth results from the simulation are presented in more detail in Figure 8.3. In the figure, the cumulative bandwidth is plotted against the number of
browsing sessions, ordered according to decreasing length of session. The right-hand end of the graph corresponds to the bandwidth results in Table 8.3. At the left end of the graph, long browsing sessions enable the ONE-PRIOR and the ALL-PRIOR methods to perform significantly better than the other three methods. As the length of the browsing sessions decrease, this advantage slowly evaporates. Furthermore, when the short browsing sessions are introduced, the cumulative bandwidth of TRANSMITTED-DICTIONARY rises sharply, and the cost of sending the priming dictionary outweighs the benefits obtained from using it.

8.5 Summary

This chapter has shown how the order users visit web pages combined with a compression system such as GZIP can achieve small additional reductions in bandwidth requirements. This idea was demonstrated through a simulation with a web server and its corresponding access logs. Additional experiments showed how a static dictionary of the most frequently occurring words on the web server also improved compression, but only if the dictionary could be assumed to be static, and available free of charge. Ultimately, the high cost of transmitting the dictionary in the many short browsing sessions was too high to justify such an approach as a general solution. The simulation has also demonstrated that GZIP
Compression of HTML Documents

alone is a good technique which reduces bandwidth requirements to 27.9%.

The experiments have also demonstrated the tension that exists between bandwidth, processing time, and storage. Bandwidth can be reduced by priming the compression system with the previously requested file. But the extra latency in compressing each file as it is requested may be unacceptable to users, who already wait impatiently during network delays. Disk space can be used to alleviate this problem by compressing all files beforehand. However, if HTML files are compressed in the context of even just one previously requested file, then a quadratic number of file variants need to be maintained. The large number of document versions is required because a user can arrive at a HTML file through any other file on the web site, or by manually entering its location in the web browser. A possible compromise would be to maintain a cache of recently requested documents, in their various compressed forms, which in the simulation was assumed to be one week. Nevertheless, it seems unlikely that all three factors can be simultaneously reduced.

Future extensions to this work, include the possible development of a compression mechanism specifically for HTML documents that offers improved compression ratios. However, unlike syntax-directed compression systems (for example, Katajainen et al. [1986]), such a system must be capable of accepting syntactically incorrect data, as HTML documents are often poorly structured and only rarely are error free. Also, while experimental results have shown that priming with a standard dictionary is problematic due to the added cost of sending it, it may be possible to separate a web server into logical domains and then to create a unique dictionary for each domain.
Chapter 9

Summary

This thesis details the compromises required in order to satisfy the two opposing tasks of compression and phrase browsing. While a compression system reduces the size of a document by removing any redundancies, a retrieval system generally augments redundant information to the document in order to facilitate efficient access. These two document management methods usually exist in isolation; but this thesis considers the costs and benefits of combining them. Throughout, the compromise between retrieval and compression has translated into a balance between disk space and retrieval time.

Chapter 2 began with an overview of the foundations of text compression, and continued by listing three complete compression systems (GZIP, BZIP2, and PPMd) and two coding mechanisms (arithmetic coding and Huffman coding) as examples. In Chapter 3, a fourth compression program called RE-PAIR [Larsson and Moffat, 2000] was introduced. Character-based RE-PAIR, as it was later called, reduces the length of a document (or message) by recursively replacing the most frequently occurring pair of symbols with a new symbol until no pair of adjacent symbols exists twice or more in the message. RE-PAIR produces a phrase hierarchy which is encoded using chiastic slide and interpolative coding, and a sequence whose coding is varied.

The RE-PAIR system forms the basis of the retrieval system in this thesis because of two other properties. As a dictionary-based compression mechanism, the decoding time is fast, an important concern for interactive retrieval systems. Also, the phrase hierarchy is both compact and rich with information. After encoding, it occupies only 0.223 bpc compared
Summary

to the original data, but contains information about the characters in the document and the relationships they form to make partial or complete words and phrases. A retrieval system based on browsing these symbols has the advantage of not requiring any of the auxiliary data structures to be permanently stored. Chapter 3 concluded by noting three drawbacks to character-based RE-PAIR, which are addressed by the enhancements that form the main body of this thesis.

Chapter 4 added a pre-processing step to RE-PAIR called PRE-PAIR. The PRE-PAIR system separates a message into two alternating streams of word tokens and non-word tokens. Each of these streams is accompanied by a lexicon. Word tokens are also case folded and stemmed in the interest of reducing the size of the word lexicons, resulting in a total of seven streams. Pairing rules are augmented to RE-PAIR to prevent phrases from straddling punctuation marks. Combining all of these additions results in a system called punctuation-aligned RE-PAIR. The phrase hierarchy, coupled with the word lexicon, now contains the words that occur in the document and the relationships they form to make phrases.

In order to tackle the problem of memory usage, Chapter 5 first described how Larsson and Moffat [2000] segments a large message into blocks. Then, RE-MERGE was described as a method of combining compressed blocks in phases. An extension to RE-MERGE which allows text to be appended to a previously compressed document is also presented.

Chapter 6 looks at alternative encoding mechanisms for the reduced word sequence and the three modifier streams (case folding, stemming, and non-words). The reduced word sequence is encoded with a byte-aligned coder called RE-VIEW, while the modifiers are encoded with an indexed variant of SHUFF. The decision to switch to these methods from SHUFF was based on improved access times rather than compression effectiveness.

Attention shifted from compression to phrase browsing in Chapter 7. The RE-PHINE system enables the symbols that make up the phrase hierarchy to be browsed, and the contexts in which they occur to be retrieved. Symbols can be extended, or broken into their components. Once a symbol of interest has been isolated, the reduced word sequence is searched for the symbol in question, and the contexts are displayed one-by-one. Further
refinement through the inclusion of case folding, stemming, and intervening non-word characters is also possible.

Chapter 8 takes a detour from the primary theme of the thesis by looking at the issue of web page delivery and compression.

9.1 Putting it all together

The systems mentioned from Chapter 3 to Chapter 7 work in concert to provide document storage and browsing. Together, they are nicknamed RE-STORE, and are summarised in Figure 9.1. In the figure, boxes indicate the major components that comprise RE-STORE, while lines between them indicate the flow of data. On the right side of the figure, the matching chapter of each system is shown.

As the pieces of RE-STORE came together, the loss in compression effectiveness in order to satisfy browsing has been noted throughout the thesis. Overall, the compression effectiveness of the NEWS document has worsened from 1.674 bpc with PPMD to 2.523 bpc with punctuation-aligned RE-PAIR. Similarly, by building a retrieval system on top of a compression mechanism, drawbacks become apparent. For example, the browser is unable to accurately judge the usefulness of symbols in the phrase hierarchy, and resorts to displaying all of them instead. This is the equilibrium point between compression and browsing for RE-STORE.

Opportunities exist for future work that may end up bringing compression and browsing closer together. For example, as Figure 4.5 on page 103 shows, the modifiers add almost 1.0 bpc to the compressed message when encoded independently with SUFF. Compression levels would probably improve if these three streams were compressed together, so that the context for any one stream is formed from the two remaining ones. While compression levels closer to character-based RE-PAIR might then be attained, the effect on browsing would need to be re-assessed.

Section 7.5 gave few directions for possible improvements to phrase browsing. One suggestion is filtering primitives and phrases from RE-PHNE so that only useful ones are shown. Alternatively, a ranking scheme can be adopted which shows interesting symbols at
the top of a RE-PHINE window. Another important direction for RE-STORE is to modify the appending phase of Re-Merge (Section 5.7 on page 150) to allow novel words to be handled. Then, documents larger than 1,000 MiB can be processed.

9.2 Towards the Future

Compression and information retrieval are both well-studied areas in the field of document management. However, the successful coupling of these two areas offers new opportunities
for future work. Effective storage and improved accessibility ensure that collected data can accumulate while still remaining useful.

But is there room in information retrieval for phrase browsing? This question can be divided into two parts. First, is there a need for phrase browsing? Also, are there challenges in developing phrase browsing?

A closer look at how people approach new books will help in addressing the first question. Indexes and table of contents help people expedite their search for information between book covers. However, the words and phrases that lie on these pages also allow the reader to browse the book, essentially for free. No extra amount of work needs to be expended by the author or the publisher to facilitate browsing. If electronic documents are extensions of books, then it is expected that the tools already available for the latter will be useful for the former.

So, the problem for phrase browsing is to adapt concepts from printed matter to electronic media. Solutions employed by index-based systems cannot be replicated for phrase browsing. While metrics such as precision and recall have been established for querying, the effectiveness of phrase browsing is more difficult to measure. Precision and recall indicate how well a system has performed and how it compares to other systems. While phrase browsing can be assessed through user studies, precise work in this area is notoriously difficult. Also, the techniques and data structures for index-based systems differ from what is needed in a phrase browsing system.

In an ideal situation, an information retrieval system should provide a selection of tools which can appeal to a variety of users. Index-based querying and phrase browsing are two possible tools, and just as this thesis has looked at combining compression with phrase browsing, another area of work may lie in combining multiple information retrieval techniques into a single, succinct system.

Not only are tools merging to form more complex systems, but the format of data is changing as well. While this thesis has presented techniques for browsing compressed document collections in SGML markup, the documents have been treated like flat text. As more documents are published electronically with SGML or XML markup, compressing
Summary

and browsing them using the tags is becoming another important direction for the future. The structural information in these document formats might improve compression levels and allow users to selectively browse parts of the documents.
# Appendix A

## Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha, \beta, \delta, \gamma$</td>
<td>A symbol</td>
</tr>
<tr>
<td>$</td>
<td>\alpha</td>
</tr>
<tr>
<td>@, †</td>
<td>A replacement symbol introduced by RE-PAIR</td>
</tr>
<tr>
<td>ω</td>
<td>String of symbols</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>Alphabet of message (for RE-PAIR, the primitives)</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Set of new symbols created by RE-PAIR (phrases)</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>.</td>
<td>A space character</td>
</tr>
<tr>
<td>$\mathcal{P},</td>
<td>\mathcal{P}</td>
</tr>
<tr>
<td>$\mathcal{P}_1, \mathcal{P}_2$</td>
<td>Initial phrase hierarchies before block merging</td>
</tr>
<tr>
<td>$\mathcal{P}'$</td>
<td>Final phrase hierarchy after block merging</td>
</tr>
<tr>
<td>$\mathcal{S},</td>
<td>\mathcal{S}</td>
</tr>
<tr>
<td>$\mathcal{S}_1, \mathcal{S}_2$</td>
<td>Initial sequences before block merging</td>
</tr>
<tr>
<td>$\mathcal{S}'$</td>
<td>Final sequences after block merging</td>
</tr>
<tr>
<td>$C$</td>
<td>Set of symbols searched for in the compressed word sequence</td>
</tr>
<tr>
<td>$\mathcal{R}$</td>
<td>Set of results returned from search in the compressed word sequence</td>
</tr>
</tbody>
</table>

### SI Prefixes

- 1 KiB = 1 kibyte = $2^{10}$ bytes = 1,024 bytes
- 1 MiB = 1 mebibyte = $2^{20}$ bytes = 1,048,576 bytes
- 1 GiB = 1 gibibyte = $2^{30}$ bytes = 1,073,741,824 bytes
Appendix B

Porter Stemming Algorithm

The Porter stemming algorithm was originally described by Porter [1980] and later reprinted in 1997. The algorithm is used in the word-based and punctuation-aligned RE-PAIR variants to stem words to their root form by removing word suffixes. Chapter 4 provides details of the different methods of selecting phrases for RE-PAIR and how the word-based pre-processor, PRE-PAIR, incorporates the algorithm. This appendix provides information specific to the stemming algorithm, with emphasis on the modifiers for stemming reversal.

The version of the algorithm employed by PRE-PAIR transforms an English word into its root form by applying a sequence of six independent steps for suffix removal. Step 0 is the first step and consists of a single action. Step 1 is composed of multiple sub-steps (1a to 1c), unlike Steps 2, 3, and 4. And step 5 consists of two sub-steps (5a and 5b), applied one after the other. Each step consists of rules which either remove characters from the end of the word, or change characters at the end of the word, or do a combination of both. A rule is applied if a certain set of conditions is met. Some possible conditions include a match on the suffix, or a minimum number of consonant and vowel combinations before the stem.

Some changes to the traditional implementation of the Porter stemming algorithm were necessary for the implementation found in PRE-PAIR. The differences are as follows:

1. If a word ends in “logi”, then the final “i” is removed. (New rule for step 2)
2. If a word ends in “bli”, then the final “i” is changed to an “e”. (Updated rule for step 2.)

3. Words of length 3 characters or less, or which are SGML tags that end with a “>” character are not processed.

4. Seven contractions which include the apostrophe are removed in step 0.

The first two changes were suggested by Porter on the web page created for the algorithm, located at http://www.tartarus.org/~martin/PorterStemmer/. The last two changes were added specifically for PRE-PAIR. The third change prevents short words from being stemmed further and improves overall efficiency. In an SGML document, word tokens which end with a “>” character would not be stemmed according to the algorithm. The modification to the algorithm allows PRE-PAIR to identify tags immediately, without attempting to stem them. The last change introduces a step 0 to the stemming mechanism in PRE-PAIR for the removal of certain suffixes which include an apostrophe. The seven contractions identified by step 0 are some of the most common ones in English [Flick and Millward, 1993, pg. 217]. In particular, they include contractions where a verb occurs with the word “not”, or when a pronoun is followed by an auxiliary verb. In other cases not covered by these rules, the apostrophe is treated as a non-word during parsing. For example, “o’clock” contains two word tokens, with the first one being the letter “o” by itself. As an example of the apostrophe handling, the word “king’s” is first treated as a single word token. Step 0 removes the apostrophe and the “s”, since the “s” is too short to be useful for phrase browsing. The alternative would result in “king” and “s” being processed as two separate word tokens, to the detriment of any subsequent phrase searching.

In this investigation, stemming is required for compression, and must be reversible. To achieve reversibility, a modifier is stored along with the stemmed word to indicate which rules must be applied to invert the stemming process. While the stemming process requires knowledge about the length of the word token and the characters which precede the stem, the inverse is more mechanical and applies the steps from 5b to 0 without checking the
Porter Stemming Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of Rules</th>
<th>Number of Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>5b</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5a</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>1c</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1b</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1a</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>67</td>
<td>23</td>
</tr>
</tbody>
</table>

Table B.1: The steps in PRE-PAIR’s version of the Porter stemming algorithm. The steps are listed in the order they are used when decoding. The second column lists the number of rules in each step. The number of bits required to express which rule was used in that step is shown in the third column. In total, a 23-bit modifier is required for each word token to indicate how to reverse the stemming.

Moreover, in the algorithm, several rules within a step remove the same suffix, but based on different conditions on the characters preceding the suffix. Since the inverse of all of these rules add the same suffix back, they can be combined into a single modifier. The first rule of every step indicates that the step was not used. Only one rule is applied per step.

Table B.1 lists the steps in the order in which they are applied when decoding. Since each step is independent of other steps, a modifier for a word token is 23 bits in length. The table also lists the number of rules per step. In a few cases, the number of possible rules that can be represented by the bits allocated exceeds the actual number of rules. For example, 5 bits for step 4 can account for 32 rules. While the size of the modifier could be reduced by merging steps, the steps were separated due to encoding and decoding efficiency reasons. Chapter 6 takes a closer look at the distribution of stemming modifiers when coding the stream of stemming modifiers is examined.

The remainder of this appendix shows the decoding rules for each step. Each rule applies changes to the current word, $\omega$. In the following tables, most rules appear in the following form: $\omega e \rightarrow \omega$ional. This means that the letter “e” which ends the word is replaced with “ional”. Other rules may simply append the suffix on to the word, without
Porter Stemming Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Value</th>
<th>Word (ω)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5b</td>
<td>0</td>
<td>commun</td>
</tr>
<tr>
<td>5a</td>
<td>0</td>
<td>commun</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>communic</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>communicate</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>communication</td>
</tr>
<tr>
<td>1c</td>
<td>0</td>
<td>communication</td>
</tr>
<tr>
<td>1b</td>
<td>0</td>
<td>communication</td>
</tr>
<tr>
<td>1a</td>
<td>2</td>
<td>communications</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>communications</td>
</tr>
</tbody>
</table>

Table B.2: Example of transforming the word “commun” back to the original word “communications” using its corresponding modifier, starting from step 5b. No changes are applied to the word whenever rule 0 is found.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to ω</td>
</tr>
<tr>
<td>1</td>
<td>Double final letter in ω</td>
</tr>
</tbody>
</table>

(b) Step 5b requires 1 bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to ω</td>
</tr>
<tr>
<td>1</td>
<td>ω → ωe</td>
</tr>
</tbody>
</table>

(a) Step 5a requires 1 bit.

Figure B.1: Porter stemming algorithm – step 5.

any prior changes to the word.

As an example of how stemming is reversed, suppose the initial stemmed word token is “commun” and its corresponding modifier is: 0, (2, 0, 0), 7, 1, 5, (0, 0). Step 0 is represented by the first “0”, and parentheses have been inserted around the composite steps 1 and 5 to improve legibility. Table B.2 demonstrates how the modifier is used to recover the original word, “communications”, starting from step 5b.
### Porter Stemming Algorithm

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to (\omega)</td>
<td>10</td>
<td>(\omega \rightarrow \omega)ment</td>
</tr>
<tr>
<td>1</td>
<td>(\omega \rightarrow \omega)al</td>
<td>11</td>
<td>(\omega \rightarrow \omega)ent</td>
</tr>
<tr>
<td>2</td>
<td>(\omega \rightarrow \omega)ance</td>
<td>12</td>
<td>(\omega \rightarrow \omega)ion</td>
</tr>
<tr>
<td>3</td>
<td>(\omega \rightarrow \omega)ence</td>
<td>13</td>
<td>(\omega \rightarrow \omega)ou</td>
</tr>
<tr>
<td>4</td>
<td>(\omega \rightarrow \omega)er</td>
<td>14</td>
<td>(\omega \rightarrow \omega)ism</td>
</tr>
<tr>
<td>5</td>
<td>(\omega \rightarrow \omega)ic</td>
<td>15</td>
<td>(\omega \rightarrow \omega)ate</td>
</tr>
<tr>
<td>6</td>
<td>(\omega \rightarrow \omega)able</td>
<td>16</td>
<td>(\omega \rightarrow \omega)iti</td>
</tr>
<tr>
<td>7</td>
<td>(\omega \rightarrow \omega)ible</td>
<td>17</td>
<td>(\omega \rightarrow \omega)ous</td>
</tr>
<tr>
<td>8</td>
<td>(\omega \rightarrow \omega)ant</td>
<td>18</td>
<td>(\omega \rightarrow \omega)ive</td>
</tr>
<tr>
<td>9</td>
<td>(\omega \rightarrow \omega)ement</td>
<td>19</td>
<td>(\omega \rightarrow \omega)ize</td>
</tr>
</tbody>
</table>

Step 4 requires 5 bits.

Figure B.2: Porter stemming algorithm – step 4.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to (\omega)</td>
<td>0</td>
<td>No change to (\omega)</td>
</tr>
<tr>
<td>1</td>
<td>(\omega \rightarrow \omega)ate</td>
<td>1</td>
<td>(\omega e \rightarrow \omega)nial</td>
</tr>
<tr>
<td>2</td>
<td>(\omega \rightarrow \omega)ative</td>
<td>2</td>
<td>(\omega \rightarrow \omega)al</td>
</tr>
<tr>
<td>3</td>
<td>(\omega \rightarrow \omega)ize</td>
<td>3</td>
<td>(\omega e \rightarrow \omega i)</td>
</tr>
<tr>
<td>4</td>
<td>(\omega \rightarrow \omega)iti</td>
<td>4</td>
<td>(\omega \rightarrow \omega r)</td>
</tr>
<tr>
<td>5</td>
<td>(\omega \rightarrow \omega)al</td>
<td>5</td>
<td>(\omega \rightarrow \omega l)i</td>
</tr>
<tr>
<td>6</td>
<td>(\omega \rightarrow \omega)ful</td>
<td>6</td>
<td>(\omega e \rightarrow \omega)ation</td>
</tr>
<tr>
<td>7</td>
<td>(\omega \rightarrow \omega)ness</td>
<td>7</td>
<td>(\omega e \rightarrow \omega)on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>(\omega e \rightarrow \omega r)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>(\omega \rightarrow \omega)ism</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>(\omega \rightarrow \omega)ness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>(\omega \rightarrow \omega)iti</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>(\omega e \rightarrow \omega)iti</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>(\omega l e \rightarrow \omega)il(\omega)iti</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>(\omega \rightarrow \omega i)</td>
</tr>
</tbody>
</table>

(a) Step 3 requires 3 bits.  
(b) Step 2 requires 4 bits.

Figure B.3: Porter stemming algorithm – steps 3 and 2.
Porter Stemming Algorithm

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to $\omega$</td>
</tr>
<tr>
<td>1</td>
<td>$\omega \rightarrow \omega y$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to $\omega$</td>
</tr>
<tr>
<td>1</td>
<td>$\omega \rightarrow \omega d$</td>
</tr>
<tr>
<td>2</td>
<td>$\omega \rightarrow \omega w$</td>
</tr>
<tr>
<td>3</td>
<td>Double the final letter in $\omega$ and then apply $\omega \rightarrow \omega ed$</td>
</tr>
<tr>
<td>4</td>
<td>$\omega \rightarrow \omega ing$</td>
</tr>
<tr>
<td>5</td>
<td>$\omega e \rightarrow \omega ing$</td>
</tr>
<tr>
<td>6</td>
<td>Double the final letter in $\omega$ and then apply $\omega \rightarrow \omega ing$</td>
</tr>
</tbody>
</table>

(c) Step 1c requires 1 bit.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to $\omega$</td>
</tr>
<tr>
<td>1</td>
<td>$\omega \rightarrow \omega es$</td>
</tr>
<tr>
<td>2</td>
<td>$\omega \rightarrow \omega s$</td>
</tr>
</tbody>
</table>

(a) Step 1a requires 2 bits.

<table>
<thead>
<tr>
<th>Value</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No change to $\omega$</td>
</tr>
<tr>
<td>1</td>
<td>$\omega \rightarrow \omega d$</td>
</tr>
<tr>
<td>2</td>
<td>$\omega \rightarrow \omega m$</td>
</tr>
<tr>
<td>3</td>
<td>$\omega \rightarrow \omega s$</td>
</tr>
<tr>
<td>4</td>
<td>$\omega \rightarrow \omega t$</td>
</tr>
<tr>
<td>5</td>
<td>$\omega \rightarrow \omega ll$</td>
</tr>
<tr>
<td>6</td>
<td>$\omega \rightarrow \omega re$</td>
</tr>
<tr>
<td>7</td>
<td>$\omega \rightarrow \omega ve$</td>
</tr>
</tbody>
</table>

Step 0 requires 3 bits.

Figure B.4: Porter stemming algorithm – step 1.

Figure B.5: Porter stemming algorithm – step 0.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


250


W. B. Frakes. Introduction to information storage and retrieval systems. In Frakes and Baeza-Yates [1992], pages 1-12. [84]

BIBLIOGRAPHY


Free Software Foundation. ZGREP (wrapper for GREP program), 2001. Adapted from a version by Charles Levert <charles@comm.polymtl.ca>. [42]


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


A. I. Wirth and A. Moffat. Can we do without ranks in Burrows Wheeler transform compression? In Storer and Cohn [2001], pages 419–428. [29]


BIBLIOGRAPHY


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Wan, Raymond

Title:
Browsing and searching compressed documents

Date:
2003-12

Citation:

Publication Status:
Unpublished

Persistent Link:
http://hdl.handle.net/11343/38920

File Description:
Browsing and searching compressed documents

Terms and Conditions:
Terms and Conditions: Copyright in works deposited in Minerva Access is retained by the copyright owner. The work may not be altered without permission from the copyright owner. Readers may only download, print and save electronic copies of whole works for their own personal non-commercial use. Any use that exceeds these limits requires permission from the copyright owner. Attribution is essential when quoting or paraphrasing from these works.