Compression of Large DNA Databases

Author
Shanika Sewwandini Kuruppu

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

Department of Computer Science and Software Engineering
Melbourne School of Engineering
The University of Melbourne

January, 2012
Abstract

The thesis explores algorithms to efficiently store and access repetitive DNA sequence collections produced by large-scale genome sequencing projects. First, existing general-purpose and DNA compression algorithms are evaluated for their suitability for compressing large collections of DNA sequences. Then two novel algorithms for compressing large collections of DNA sequences are introduced. The first algorithm is COMRAD, which is a disk-based dictionary compression algorithm that iteratively detects repetitions that occur across multiple sequences, and substitutes them with non-terminals. The method showed that repeats can be feasibly detected across multiple sequences for relatively large collections, while preserving sequence boundaries. The second algorithm is RLZ, which compresses highly similar sequence collections using a simple LZ77 parsing of each sequence with respect to a sequence chosen to be the reference. RLZ was also extended to conduct non-greedy LZ77 parsing, and with the combination of a few other optimisations, the algorithm indirectly detects and encodes approximate matches. RLZ is memory efficient, and is one of the fastest DNA sequence compression algorithms existing to date, both in terms of the compression and decompression speed. Both COMRAD and RLZ can compress sequence collections of fully assembled chromosomes and genomes, as well as sets of contigs and reads. Both algorithms also support individual sequence extraction and random access queries on compressed sets. RLZ was also extended to a full self-index by enabling substring searching on compressed sets with some limitations on detecting substrings. Since the effectiveness of RLZ compression depends on the reference sequence chosen for a collection, techniques for constructing reference sequences that better represent the collection were also explored. The results showed that using a reference sequence constructed with repeats detected by dictionary compressors leads to significant improvements in the compressed sizes produced by RLZ.
Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Shanika Sewwandini Kuruppu
Department of Computer Science and Software Engineering
The University of Melbourne
January, 2012
Preface

The contributions made in Chapters 5–8 of this thesis appear in the following publications of which I was the primary author. The specific contributions are outlined at the beginning of the relevant chapters.


Acknowledgements

First and foremost I would like to thank my primary supervisor Prof Justin Zobel, who helped me to choose an interesting and timely research topic. He was the most prominent figure in guiding me to complete my candidature, and his advice, stern words and attention to detail were invaluable.

I would also like to thank my secondary supervisor Dr Simon Puglisi for fuelling my interest in algorithms and data structures, and for guiding me to appreciate research. I thoroughly enjoyed our whiteboard algorithm discussion sessions. I also wish to thank my two other secondary supervisors Dr Thomas Conway and Dr Bryan Beresford-Smith, who guided my research direction earlier in my candidature. I am grateful to all my supervisors for proofreading the thesis and providing me with valuable feedback. I would also like to thank my advisory committee consisting of Prof Alistair Moffat and Dr Linda Stern for ensuring the timely completion of my candidature and for their valuable advice at progress meetings.

I would also like to thank my handsome partner Matt Giuca for his support during my candidature, with proofreading my paper submissions and thesis, helping me work through incomprehensible compiler errors and software bugs, and for being a wonderful office- and house-mate. I would also like to thank the Giuca family for their support and my parents for teaching me the value of education.

I would also like to thank the NICTA genomics group and the CSSE department for supporting my research. Finally, I would like to thank NICTA for funding my candidature, especially the conference travels, which allowed me to present my research, broaden my knowledge and meet some talented researchers in the field.
# Contents

## 1 Introduction

1.1 Contributions ............................................. 3

1.2 Thesis structure .......................................... 4

## 2 DNA Background

2.1 Genomic data .............................................. 9

2.1.1 DNA and proteins ....................................... 9

2.1.2 Mutations in DNA ....................................... 11

2.1.3 Evolutionary conservation .............................. 14

2.1.4 Repetition in DNA ...................................... 16

2.1.5 Genomic datasets ....................................... 19

2.2 DNA Sequencing ........................................... 21

2.2.1 Sequencing methods .................................... 21

2.2.2 Sequence assembly ..................................... 22

2.2.3 Genome sequencing projects ......................... 23

2.2.4 Applications of sequenced DNA .................... 25

2.3 Chapter Summary .......................................... 27

## 3 Compression Background

3.1 Strings and string data structures ....................... 30

3.1.1 String notation ........................................ 30

3.1.2 Queries on strings .................................... 30

3.1.3 Suffix trees and suffix arrays ....................... 31

3.1.4 Compact data structures .............................. 33

3.2 General-purpose compression ......................... 34

3.2.1 Compression performance ............................ 36

3.2.2 Dictionary compression ............................... 38

  Adaptive dictionary compression ...................... 39

  Semi-static dictionary compression ................... 44

3.2.3 Statistical compression ............................... 47

3.2.4 Burrows-Wheeler transform ......................... 48

3.2.5 Encoding techniques ................................. 50
### CONTENTS

3.3 DNA compression .................................................. 55
  3.3.1 Exact matching ............................................. 56
  3.3.2 Approximate matching ..................................... 59
  3.3.3 Compressing variations .................................. 63
3.4 Self-indexes ..................................................... 65
  3.4.1 Burrows-Wheeler index .................................. 66
  3.4.2 Compressed suffix arrays ................................. 67
  3.4.3 Lempel-Ziv index ......................................... 68
3.5 Chapter Summary ............................................... 71

4 Existing compression algorithm evaluation .......................... 73
  4.1 Test data ...................................................... 73
    4.1.1 Single sequence dataset ............................... 74
    4.1.2 Repetitive dataset ..................................... 74
    4.1.3 Dataset repeat properties .............................. 76
  4.2 Testing methodology .......................................... 78
    4.2.1 Measuring performance .................................. 78
      Compressed size ........................................... 78
      Compression and decompression time ...................... 79
      Memory usage ............................................. 79
  4.3 General-purpose compression algorithm evaluation ............ 80
    4.3.1 Input parameters ........................................ 80
    4.3.2 Results ................................................ 81
      Compressed size .......................................... 81
      Compression and decompression speed ..................... 84
      Memory usage ............................................. 85
  4.4 DNA compression algorithm evaluation ........................ 88
    4.4.1 Algorithm parameters .................................. 88
    4.4.2 Results ................................................ 90
      Compressed size .......................................... 90
      Compression and decompression speed ..................... 90
      Memory usage ............................................. 92
  4.5 Discussion .................................................. 94

5 Efficient genome storage with COMRAD ............................. 97
  5.1 Modifications to RAY ........................................ 98
  5.2 COMRAD compression .......................................... 101
    5.2.1 Algorithm parameters .................................. 101
    5.2.2 Compression algorithm ................................. 103
    5.2.3 Decompression algorithm ............................... 110
5.2.4 Time and space complexity .................................................. 113
5.2.5 Experimental evaluation .................................................... 116
  Test data and environment ..................................................... 116
  Results for repetitive sequence collections .............................. 117
  Results for single sequences ................................................. 121
5.2.6 Effects of changing parameters .......................................... 123
5.2.7 Discussion ........................................................................ 127
5.3 The display() query .............................................................. 130
  5.3.1 Algorithm ..................................................................... 130
5.3.2 Time and space complexity ................................................ 134
5.3.3 Experimental evaluation .................................................... 136
  Test data and environment ..................................................... 136
  Results .............................................................................. 137
5.3.4 Discussion ........................................................................ 137
5.4 Chapter summary ................................................................. 139

6 Efficient genome storage with RLZ ........................................... 141
6.1 Relative Lempel-Ziv (RLZ) compression ................................. 143
  6.1.1 Inputs .......................................................................... 143
  6.1.2 Compression algorithm .................................................... 144
    Relative Lempel-Ziv factorisation ........................................... 144
    Factor encoding .................................................................. 147
    Time and space complexity ................................................. 148
    Effects of mutations on factorisation .................................... 149
  6.1.3 Decompression algorithm .................................................. 152
  6.1.4 Experimental evaluation .................................................... 153
    Test data and environment ................................................... 153
    Results ............................................................................. 154
6.2 Optimised Relative Lempel-Ziv compression ............................ 157
  6.2.1 Non-greedy Lempel-Ziv factorisation ................................... 158
    Non-greedy relative Lempel-Ziv factorisation algorithm .......... 159
    Local maximum factor detection ......................................... 161
    Matching-statistics ............................................................ 163
  6.2.2 Efficient short factor encoding .......................................... 165
  6.2.3 Efficient factor position encoding ...................................... 167
  6.2.4 Experimental evaluation .................................................... 171
    Test data and environment ................................................... 171
    Results ............................................................................. 171
6.3 Discussion ............................................................................ 177
6.4 Chapter summary ................................................................. 182
7 The RLZ self-index

7.1 The display() query

7.1.1 Modifications to RLZ-compressed output

7.1.2 The display() algorithm

7.1.3 Time and space complexity

7.1.4 GDC display() query

7.1.5 Experimental evaluation

7.1.6 Discussion

7.2 Search: count() and locate() queries

7.2.1 Modifications to the RLZ index

7.2.2 The search algorithm

7.2.3 Time and space complexity

7.2.4 Experimental evaluation

7.2.5 Discussion

7.3 Chapter summary

8 Reference construction

8.1 Reference selection

8.2 Dictionary compressors for reference construction

8.2.1 Experimental evaluation

8.2.2 Discussion

8.3 Heuristics for repeat detection

8.4 Chapter summary

9 Conclusion

Bibliography

A Data and algorithm sources

B COMRAD implementation
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example DNA sequence</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Example DNA sequence with coding and template strands</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Two DNA sequences with mutations</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Two DNA sequences with an insertion and a deletion</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Large-scale mutations occurring in genomes</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Phylogenetic tree for <em>P. troglodytes</em> and species of the <em>Homo</em> genus</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of exact and approximate repeats</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>Similarity between the HBZ gene sequence of <em>G. gallus</em> and <em>H. sapiens</em></td>
<td>17</td>
</tr>
<tr>
<td>2.9</td>
<td>Similarity between HBZ gene sequence of <em>H. sapiens</em> individuals</td>
<td>17</td>
</tr>
<tr>
<td>2.10</td>
<td>Comparison of four H2A histone gene copies in the <em>H. sapiens</em> genome</td>
<td>18</td>
</tr>
<tr>
<td>2.11</td>
<td>Example FASTA sequence</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Example suffix tree</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Example suffix array</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of the Burrows-Wheeler Transform</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Example Huffman tree</td>
<td>51</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of arithmetic coding</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Use of patterns for repeat substitution</td>
<td>104</td>
</tr>
<tr>
<td>5.2</td>
<td>Input sequences and input parameters into COMRAD</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Frequency dictionary for distinct L-mers</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td>Distinct L-mers with a frequency of at least F</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>Substitution step of the COMRAD first iteration</td>
<td>107</td>
</tr>
<tr>
<td>5.6</td>
<td>Second iteration frequency dictionary</td>
<td>108</td>
</tr>
<tr>
<td>5.7</td>
<td>Substitution step of the COMRAD second iteration</td>
<td>109</td>
</tr>
<tr>
<td>5.8</td>
<td>COMRAD output</td>
<td>109</td>
</tr>
<tr>
<td>5.9</td>
<td>COMRAD decompression with standard dictionary</td>
<td>111</td>
</tr>
<tr>
<td>5.10</td>
<td>COMRAD decompression with expanded dictionary</td>
<td>112</td>
</tr>
<tr>
<td>5.11</td>
<td>Comparison of compressed sizes for DATASET-REP</td>
<td>119</td>
</tr>
<tr>
<td>5.12</td>
<td>Comparison of compressed sizes for DATASET-SIN</td>
<td>122</td>
</tr>
<tr>
<td>5.13</td>
<td>Variation in the compressed size as F parameter changes</td>
<td>124</td>
</tr>
<tr>
<td>5.14</td>
<td>Variation in the compressed size as the L parameter changes</td>
<td>125</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>5.15</td>
<td>Mapping between a sequence and the compressed sequence</td>
<td>131</td>
</tr>
<tr>
<td>5.16</td>
<td>Bit vectors indicating non-terminal start positions</td>
<td>133</td>
</tr>
<tr>
<td>6.1</td>
<td>Example set of input sequences for RLZ</td>
<td>143</td>
</tr>
<tr>
<td>6.2</td>
<td>Lempel-Ziv factors as strings, and position and length pairs</td>
<td>145</td>
</tr>
<tr>
<td>6.3</td>
<td>RLZ factorisation example</td>
<td>146</td>
</tr>
<tr>
<td>6.4</td>
<td>RLZ output</td>
<td>147</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison of standard RLZ compressed sizes to existing results</td>
<td>156</td>
</tr>
<tr>
<td>6.6</td>
<td>Look-ahead example</td>
<td>158</td>
</tr>
<tr>
<td>6.7</td>
<td>Non-greedy RLZ factorisation example</td>
<td>160</td>
</tr>
<tr>
<td>6.8</td>
<td>Local maximum factors</td>
<td>162</td>
</tr>
<tr>
<td>6.9</td>
<td>Example factors that form an alignment</td>
<td>167</td>
</tr>
<tr>
<td>6.10</td>
<td>Factor length distribution for greedy and non-greedy factorisation</td>
<td>169</td>
</tr>
<tr>
<td>6.11</td>
<td>Compressed sizes as look-ahead limits are varied</td>
<td>173</td>
</tr>
<tr>
<td>6.12</td>
<td>Compression and decompression times as look-ahead limits are varied</td>
<td>175</td>
</tr>
<tr>
<td>6.13</td>
<td>Comparison of the optimised RLZ compressed sizes to existing results</td>
<td>176</td>
</tr>
<tr>
<td>7.1</td>
<td>Primary and secondary occurrence examples</td>
<td>196</td>
</tr>
<tr>
<td>7.2</td>
<td>Example nesting level list</td>
<td>198</td>
</tr>
<tr>
<td>7.3</td>
<td>Inserting intervals into a nesting level list</td>
<td>198</td>
</tr>
<tr>
<td>8.1</td>
<td>Variation in compressed size as the reference is changed</td>
<td>213</td>
</tr>
<tr>
<td>8.2</td>
<td>RLZ factors that form alignments</td>
<td>215</td>
</tr>
<tr>
<td>8.3</td>
<td>Comparison of compressed sizes for collections with a reference</td>
<td>223</td>
</tr>
<tr>
<td>8.4</td>
<td>Comparison of compressed sizes for collections without a reference</td>
<td>224</td>
</tr>
</tbody>
</table>
**List of Tables**

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Properties of the test datasets</td>
<td>76</td>
</tr>
<tr>
<td>4.2</td>
<td>Repeat properties of the test datasets</td>
<td>77</td>
</tr>
<tr>
<td>4.3</td>
<td>Compressed size produced by general-purpose compression tools</td>
<td>83</td>
</tr>
<tr>
<td>4.4</td>
<td>Average number of bits used by general-purpose compression tools</td>
<td>84</td>
</tr>
<tr>
<td>4.5</td>
<td>Compression times taken by general-purpose compression tools</td>
<td>85</td>
</tr>
<tr>
<td>4.6</td>
<td>Decompression times taken by general-purpose compression tools</td>
<td>86</td>
</tr>
<tr>
<td>4.7</td>
<td>Maximum memory used by general-purpose compression tools</td>
<td>87</td>
</tr>
<tr>
<td>4.8</td>
<td>Compressed size produced by DNA compression tools</td>
<td>91</td>
</tr>
<tr>
<td>4.9</td>
<td>Average number of bits used by DNA compression tools</td>
<td>92</td>
</tr>
<tr>
<td>4.10</td>
<td>Compression and decompression times for DNA compression tools</td>
<td>93</td>
</tr>
<tr>
<td>4.11</td>
<td>Maximum memory used by DNA compression tools</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of test dataset properties</td>
<td>116</td>
</tr>
<tr>
<td>5.2</td>
<td>COMRAD results for collection compression</td>
<td>117</td>
</tr>
<tr>
<td>5.3</td>
<td>Variation in COMRAD results for varying mutation rates</td>
<td>121</td>
</tr>
<tr>
<td>5.4</td>
<td>COMRAD results for single-sequence compression</td>
<td>121</td>
</tr>
<tr>
<td>5.5</td>
<td><code>display()</code> query performance comparison</td>
<td>138</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of index sizes</td>
<td>138</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of test collection properties</td>
<td>154</td>
</tr>
<tr>
<td>6.2</td>
<td>RLZ results for collection compression</td>
<td>155</td>
</tr>
<tr>
<td>6.3</td>
<td>XMCompress relative compression results</td>
<td>157</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison of standard RLZ and optimised RLZ results</td>
<td>176</td>
</tr>
<tr>
<td>6.5</td>
<td>Comparison of RLZ and GDC results</td>
<td>182</td>
</tr>
<tr>
<td>7.1</td>
<td>RLZ <code>display()</code> query performance comparison</td>
<td>193</td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison of index sizes</td>
<td>193</td>
</tr>
<tr>
<td>7.3</td>
<td>RLZ search query performance comparison</td>
<td>206</td>
</tr>
<tr>
<td>7.4</td>
<td>Comparison of index sizes</td>
<td>207</td>
</tr>
<tr>
<td>7.5</td>
<td>Index construction times</td>
<td>207</td>
</tr>
<tr>
<td>8.1</td>
<td>Comparison of reference construction methods</td>
<td>222</td>
</tr>
<tr>
<td>8.2</td>
<td>Break-down of compressed sizes and compression times</td>
<td>223</td>
</tr>
</tbody>
</table>
8.3 Comparison of reference construction methods for general collections 225
8.4 Comparison with GDC reference improvement methods 227
Chapter 1

Introduction

Biomedical research is faced with ever increasing quantities of data, primarily due to the recent advancements in high-throughput sequencing technologies. The GenBank database, which consolidates DNA sequences submitted by numerous research institutes, is doubling in size approximately every 30 months [Benson et al., 2009]. A survey reported by *Nature* in 2010 estimated that the total number of human genomes sequenced by the end of 2011 will be over 30,000 [Katsnelson, 2010]. Numerous sequencing projects exist that are dedicated to sequencing genomes of various species, and sequencing DNA related to certain diseases.

Analysing the genome of an organism provides vital insights into the manner in which it functions. The survey reported by *Nature* showed that more than half the complete human genomes sequenced were for the purposes of disease analysis, with a large proportion of these disease-related genomes being cancer genomes [Katsnelson, 2010]. Analysing DNA is also important for applications such as determining how organisms evolved from their common ancestors [Heywood and McNeill, 1964], genetic engineering to create disease-resistant crops, and understanding the types of organisms found in water and soil samples to determine how they maintain the environment [Pignatelli and Moya, 2011].

The process of determining the contents of a DNA sequence is known as *sequencing*. Until recently, the sequencing process was costly and time consuming, even to sequence a short piece of DNA consisting of only a few thousand nucleotides. The recent rapid technological advancements in the field have enabled large genomes to be sequenced within a matter of days at an affordable cost. This has resulted in numerous projects to sequence the genomes of various species, individuals from the same species, and many samples of DNA sequence segments of interest. Such projects are rapidly advancing our knowledge of genetics.

Sequencing projects output large quantities of data, especially when multiple large genomes are being sequenced. A component of this output is the sequenced short DNA fragments known as *reads*, and other associated information, such as

identifiers and quality scores of the reads. The reads are then assembled to produce the final genomes. The size of the assembled set of genomes depend on the number of genomes sequenced and the size of each genome. Other information such as how the reads were ordered to construct the assembled genome, and the differences between each genome with respect to a consensus genome, may also need to be stored. Consequently, a single sequencing project outputs large quantities of data.

Given the vast quantities of data generated by sequencing projects, it is attractive to explore ways to store it efficiently. Compression tools such as, gzip [Deutsch, 1996], bzip2\(^2\) and 7-Zip,\(^3\) are used frequently to compress text, images and videos, among numerous other data types. These tools can be used to compress sequencing project data, and it is indeed the norm among researchers to compress the data with gzip or bzip2 for personal storage, and for publishing it online. However, DNA has specific properties that can be exploited, potentially, with special-purpose compression tools. For example, unlike text documents, repeated substrings within a sequence tends to be inexact and far apart. Tools such as gzip and bzip2 with their small window for repeat detection cannot exploit this DNA-specific property to achieve compression.

Grumbach and Tahi [Grumbach and Tahi, 1993] were the first to realise the potential advantages of a compression tool that is tailored to compress based on the specific properties of DNA. Since then, DNA sequence compression has been a well-researched area with many existing tools that compress the sequences to a smaller size, compared to tools such as gzip [Cao et al., 2007; Chen et al., 2000, 2002; Grumbach and Tahi, 1994; Manzini and Rastero, 2004; Rivals et al., 1996]. However, most of these tools were created when the largest available sequences were only a few kilobases (a base is a nucleotide) in size, hence they were not designed to scale for compressing gigabases to terabases of sequences. Therefore, it is necessary to explore ways to improve existing compression algorithms or introduce new algorithms for compactly storing sequenced data.

In this thesis, we focus on algorithms for compressing repetitive collections of fully or partially assembled DNA sequences. Such collections are the end result of genome sequencing projects. The sequences in many of these collections are highly similar to each other, especially if the sequences originate from the same or evolutionarily close species. These repeated substrings can be detected and exploited to compress such collections effectively. For collection compression, having the ability to detect repetitions occurring across sequences is vital.

A main drawback of most existing general-purpose and DNA compression algorithms is the inability to conduct collection compression. These tools are designed for single-file compression and stream compression. Generally, the data from sequencing projects is spread across multiple files. For example, the assembled sequences

\(^2\)bzip.org
\(^3\)www.7-zip.org
are stored in files separated on a per-genome or per-sample level. If a single file is compressed at a time, the similarity in the sequences cannot be detected and used for compression. Archiving tools such as 7-Zip can compress multiple files by detecting the similarity between the current file and existing files in the archive. However, due to the limitations on the access to earlier contents of the archive, the level of compression and the access speed for individual files depends on the order in which the files are added to the archive. Some DNA-specific compression tools can compress a sequence with respect to one other sequence, but these implementations cannot scale for large collections.

Detecting repetitions occurring across sequences also has its drawbacks. Most existing compression algorithms do not permit global repeat detection as it is costly in terms of the memory usage and compression time. However, without global repeat detection, sequence collections will not be well compressed. The larger sizes of modern sequence collections pose further challenges for global repeat detection. Memory-based algorithms cannot compress large collections, while disk-based algorithms are slow to compress and decompress. Therefore, trade-offs need to be made to balance the resource usage against the compression achieved.

When large collections need to be processed and analysed, access to individual sequences or substrings of sequences is necessary, therefore a compressed collection needs to be decompressed. There may be insufficient space to store the decompressed data, and decompressing an entire collection may not be feasible. It may be desirable to perform these queries on a compressed collection to avoid the decompression overhead. Fortunately, a new type of index known as a self-index was developed [Arroyuelo et al., 2012], that stores the original collection compressed, while the access and search queries are performed directly on the compressed data. Self-indexes have shown promising results for DNA datasets [Mäkinen et al., 2010] but modifications can still be made to improve the storage space and query speeds.

1.1 Contributions

The primary focus of this thesis is on exploring algorithms for compressing assembled DNA sequence collections. The novelty of our algorithms is in their ability to detect and exploit repetitions occurring across the sequences in a collection. Given this property, we must assume that the sequences in a collection are highly similar, either being genomes or sequences from the same or evolutionarily close species, or sequences that serve the same biological function. By detecting repetitions occurring across sequences, our algorithms are able to produce good compression results, while enabling decompression of individual sequences in the collection and access to substrings from individual sequences, without decompressing the entire collection.

We introduce two compression algorithms COMRAD and RLZ in this thesis. COMRAD is a disk-based dictionary compression algorithm that iteratively detects repetitions
occurring across multiple sequences. At each iteration, repeats occurring across the collection are identified before substitutions are made to ensure that repeats are detected globally in a manner that is independent of the sequence order. While COMRAD can compress general collections of sequences, our second algorithm RLZ is designed to compress collections, where the sequences are identical except for several variations. Such collections are typically the output of resequencing projects. More specifically, RLZ selects a single sequence from the collection as a reference and compresses each of the remaining sequences in the collection with respect to this reference. Both COMRAD and RLZ can compress sequence collections of fully assembled chromosomes and genomes, as well as sets of reads and partially assembled sequences. Both algorithms can be extended to also support individual sequence extraction and random access queries on compressed collections. We also extended the RLZ algorithm to a full self-index by enabling pattern searching on compressed collections but with certain limitations. Since the effectiveness of RLZ compression depends on the reference chosen for a collection, we also explored techniques for constructing references that better represent the collection.

1.2 Thesis structure

We begin in Chapter 2 with an introduction to DNA and proteins, which are the basic molecules on which life is based. Then mutations are discussed as they are important for understanding the types of repeats present in DNA sequences. The extent of repetition in a string determines the compressibility of the string, therefore the kind of repetition typically observed within genomes, across genomes of different species, and across genomes of the same species are then discussed. Generally, the level of repetition within genomes is lower than the repetitions occurring across sequences, which justifies our focus on the latter for compression purposes.

The motivation of this research is to explore algorithms to compress the output of large-scale sequencing projects. In Chapter 2, we briefly introduce the basic concepts of DNA sequencing, including the sequencing process, and the assembly of reads to produce assembled sequences. Examples of large-scale sequencing projects made possible by high-throughput sequencing are also discussed. The chapter concludes with a discussion of applications that rely on the fast sequencing speeds and low cost of high-throughput sequencing.

The algorithms and data structures used in the thesis are presented in Chapter 3. Data structures introduced include, suffix trees and suffix arrays, which are widely used for string searching in compression algorithms and self-indexes, and compressed bit vectors, which are used in self-indexes to efficiently implement search and access queries. A detailed discussion of compression algorithms follow. We focus on dictionary compression, since this type of compression is based on repeat detection. Dictionary compression is discussed in terms of the two types, adaptive
compression and semi-static compression. Then, several encoding techniques, such as Huffman [Huffman, 1952], Arithmetic [Rissanen, 1976] and Golomb [Golomb, 1966] coding, that are used to encode the symbols and other data in the compressed output, are discussed.

DNA-specific compression algorithms are introduced next in Chapter 3. These algorithms are based on the concepts of general-purpose compression algorithms, but are tailored for DNA-specific properties to improve the compression results beyond what can be achieved by general-purpose compression algorithms. The discussion is divided into algorithms that detect exact repeats, and those that detect approximate repeats, which are suitable for detecting repeats within sequences. We compare these algorithms in terms of their compression performance, and their likely scalability for compressing large collections. Several other algorithms that compress the variations of one genome with respect to another genome are also briefly discussed.

Finally in Chapter 3, we briefly discuss some existing self-indexes. Three types of self-indexes are discussed: self-indexes based on the Burrows-Wheeler transform, compressed suffix arrays, and Lempel-Ziv parsing. The types of indexes are discussed in terms of their implementations to illustrate the manner in which self-indexes compress the input, and implement the search and access queries.

In Chapter 4, as our first contribution, we analyse the compression performance of several publicly available general-purpose and DNA-specific compression tools, to evaluate their performance on single DNA sequences and collections of DNA sequences. We introduce the test datasets used throughout the thesis and the criteria used to compare the performance of compression algorithms, which consists of the compressed size, compression and decompression speed, and the memory use.

The general-purpose compression tools and DNA compression tools are analysed using the test datasets. The results show that for single-sequence compression, two of the frequently used general-purpose tools, gzip and bzip2, produce compressed sizes that are larger than the size of the original sequences. Tools such as 7-Zip, Sequitur [Nevill-Manning and Witten, 1997b,c], and ppmd [Cleary and Witten, 1984] produce the best results. For collection compression, dictionary compression tools like 7-Zip, Re-pair [Larsson and Moffat, 1999] and Sequitur produce the best compression results, if the individual sequences in a collection are concatenated into a single file. Even after concatenating the sequences, gzip and bzip2 are unable to compress these collections well.

Using DNA compression tools, much better results could be achieved, both for single-sequence compression and collection compression. Most algorithms in the literature do not have publicly available implementations, and most available implementations do not scale to compress sequences of even a few megabases. Only the tools dna-x [Manzini and Rastero, 2004] and XMCompress were able to compress most of the sequences and collections in our test datasets. Both dna-x and XMCompress out-perform all general-purpose tools for single-sequence compression.
in terms of the compressed size, while XMCompress out-perform 7-Zip for collection compression, which in turn out-performed dna-x.

Several issues were evident from the experiments in Chapter 4. The two most frequently used compression tools by researchers for compressing DNA sequences, gzip and bzip2, are far from ideal for compressing DNA sequences. DNA compression tools such as dna-x and XMCompress are better for this purpose, but are slow, and large collections must be compressed in blocks. General-purpose compression tools such as Re-pair and Sequitur have faster compression speeds than these DNA compression tools, but large collections also must be compressed in blocks. The evaluation highlights some improvements that can be made to compression algorithms to enable collection compression.

The first technical contribution made in this thesis is the COMRAD DNA compression algorithm presented in Chapter 5. COMRAD compresses a collection of sequences over several iterations, ensuring that repeats of at least a certain minimum length are detected and compressed. At each iteration, repeat detection and compression are conducted over two separate passes of the collection to ensure that the repeats occurring across sequences are identified before compression decisions are made. COMRAD is based on RAY [Cannane and Williams, 2001], which substitutes the occurrences of one or more distinct symbol pairs at each iteration with a non-terminal symbol. First we present the COMRAD compression and decompression algorithms, then we experimentally verify the performance of our implementation of the algorithm. COMRAD produce reasonable compression results, but it does not out-perform tools such as 7-Zip, dna-x, and XMCompress. We believe that better results can be achieved by improving the encoding techniques. The results show that, for collections of several gigabases, global repeat detection across multiple sequences is feasible. In terms of the compression speed, COMRAD is slower than the general-purpose tools. In terms of the decompression speed, COMRAD compares well. The drawback of COMRAD is the high memory use resulting from the semi-static approach to detection of global repetitions, hence further improvements need to be made to enable terabase-sized collections to be compressed.

The second contribution, also in Chapter 5 is the addition of the substring extraction feature to COMRAD-compressed collections. The modifications required to the compressed output are described, followed by the algorithm for extracting substrings from compressed collections. The results for comparing our implementation of the query with that of existing self-index implementations show that the COMRAD extraction speed is slower than that of LZ-End but is faster than that of RLCSA. However, COMRAD has faster extraction speeds for longer queries, compared to other tools, making it ideal for fast individual sequence extraction. The drawback is the larger compressed size required to enable this query.

In Chapter 6, we present our next contribution, the RLZ algorithm, which is designed to compress collections output by resequencing projects. Given a collection
1.2. THESIS STRUCTURE

of highly similar genomes or sequences, \textit{RLZ} selects a sequence from the collection as a reference and compresses the remaining sequences in the collection with respect to this reference using an \textit{LZ77} parsing [Ziv and Lempel, 1977]. First the compression and decompression algorithms are presented. The impact that various types of mutations occurring in each sequence has on the compressed size is also discussed. The compression and decompression performance of the algorithm is evaluated, and the results show that \textit{RLZ} is one of the fastest compression algorithms in the literature. \textit{RLZ} is also memory-efficient compared to existing algorithms. \textit{RLZ} produces good compression results, although tools such as 7-Zip, dna-x and XMCompress (in standard and relative compression modes) produce better results than \textit{RLZ}.

The standard \textit{RLZ} algorithm uses a greedy \textit{LZ77} parsing of each sequence with respect to the reference. Although good compression results are achieved with this approach, further improvements can be made to almost halve the compressed size for certain collections. We also discuss these improvements in Chapter 6. One such improvement is to use a non-greedy \textit{LZ77} parsing instead of the greedy parsing. This modification by itself results in poorer compression results; hence the techniques used to encode the compressed output also need to be modified. By differentiating the encoding technique for shorter and longer \textit{LZ77} factors, significant compression gains can be made. Another modification is based on the fact that the sequences being compressed are highly similar; hence they form alignments to the reference. Therefore, the positions at which substrings match to in the reference are predictable, hence such factors can be encoded more efficiently. The version of the \textit{RLZ} algorithm with these improvements is identified as optimised-\textit{RLZ} and we experimentally verify the consequences of making each of these modifications, and the effects of combining these modifications. Using the combination of all three modifications, for certain highly-repetitive sequence collections, the compressed results almost halved. The drawbacks of these improvements are the increased compression time and memory use. The \textit{RLZ} decompression speed tends to improve as a result of these changes.

The \textit{RLZ} compression algorithm described in Chapter 6 also supports individual sequence decompression. The compressed output of \textit{RLZ} can be modified to create an \textit{RLZ} self-index that supports substring extraction and search queries. These modifications and the algorithms to perform self-index queries are discussed in Chapter 7. The most trivial query to implement is the substring extraction query. The experimental evaluation of the \textit{RLZ} query shows that it has the fastest extraction speed compared to existing implementations that we compared to. The index size is also smaller than those of existing implementations like \textit{LZ-End} [Kreft and Navarro, 2010, 2011], \textit{RLCSA} [Mäkinen et al., 2009; Mäkinen et al., 2010] and \textit{COMRAD}.

Next we add further data structures to the compressed \textit{RLZ} output so that substring searching can be supported by the \textit{RLZ} self-index. Due to the difficulties associated with searching for substring occurrences across \textit{LZ77} phrase boundaries, the addition of the search query was non-trivial. The \textit{RLZ} self-index has the lim-
CHAPTER 1. INTRODUCTION

The limitation of not being able to detect patterns occurring across more than two LZ77 phrases, an issue that will be resolved in future research. The experimental evaluation of the RLZ search queries show that the query performance is significantly slower than the query performance of existing self-indexes. However, the RLZ index sizes are comparable to the other index sizes. Constructing an RLZ index is also faster than constructing the other indexes. We present some suggestions for further modifications that can be made to improve the query performance.

One of the major drawbacks of the RLZ algorithm, and relative compression in general, is the selection of an appropriate reference. Certain collections may specify the sequence to be used as a reference, but in general, there is no prior knowledge of which sequence should be used as the reference. In Chapter 8, we show the consequences of choosing a bad reference, then discuss the difficulties of selecting a good reference and managing multiple references. A good reference must contain repeats that are present in the entire collection to achieve good compression. Therefore we propose a technique for artificially constructing a reference from the repeats detected by a dictionary compression tool for the collection. We chose several dictionary compression tools to construct references, and we experimentally verify the suitability of these tools for reference construction. The results show that using semi-static dictionary compression tools to construct a reference produces the best compression results. We also compare our results to an improved version of the RLZ algorithm, GDC [Deorowicz and Grabowski, 2011], that also explores ways to improve the quality of the reference. Our approach achieves better results than most techniques explored by Deorowicz and Grabowski [2011]. Using two compression algorithms, one to construct a reference, and the other to compress the collection, cannot scale for large collections. Therefore, we briefly explore some heuristics for reference construction.

Finally, in Chapter 9 we summarise the contributions made in this thesis and discuss several research ideas to be explored further relating to the management of large collections of DNA sequences.

The algorithms we introduce in this thesis contribute to easing the storage and access issues associated with managing multi-gigabase collections. We expect that these algorithms can form the basis for algorithms that manage multi-terabase to petabase collections of DNA sequences.
Chapter 2

DNA Background

Analysing DNA sequences is important for applications in many areas. In medical research, mutations in DNA sequences are analysed to understand the causes of diseases and then to determine appropriate treatment. In genetics, detecting similarities in the DNA sequences of various species is important to better understand the evolutionary relationships between the species. In agriculture, crops can be made resistant to certain diseases through manipulation of the DNA sequences to improve the harvest. Certain other modifications may even result in more nutritious produce. These are just some examples of applications that use DNA, among many others. For many of these applications, the ability to store and access large quantities of DNA data is very important. The problem being addressed in this thesis is to explore algorithms that can store and access large DNA datasets efficiently.

Before addressing the problem, it is important to understand the properties of DNA and the origin and nature of DNA datasets. In Section 2.1, we begin with a brief introduction to DNA that defines some of the terminology used in the thesis. We also provide an overview of the functionality DNA plays in maintaining living organisms, followed by a discussion of some of the repeat properties of DNA that are interesting from the point of view of compression. In Section 2.2, we introduce DNA sequencing technologies and the recent improvements that have led to the numerous sequencing projects that are currently underway. We describe some prominent sequencing projects and the applications of high-throughput sequencing.

2.1 Genomic data

2.1.1 DNA and proteins

Deoxyribonucleic acid or DNA forms the genetic instructions for all living organisms, which allows an organism to develop, function and reproduce. DNA is a long polymer made up of building blocks known as nucleotides. There are four distinct nucleotides; Adenine (a), Cytosine (c), Guanine (g), and Thymine (t). An example DNA sequence is presented in Figure 2.1.
DNA molecules are double stranded, where the two strands are connected by hydrogen bonds. The two strands are paired such that an Adenine or Cytosine nucleotide in one strand is always paired with a Thymine or Guanine nucleotide on the other strand, respectively. Therefore, knowing the series of nucleotides on one strand allows us to infer the series of nucleotides on the opposite strand, which is an important property that allows DNA to replicate. Figure 2.2 illustrates the two strands of our example yeast DNA sequence. One of the strands is identified as the coding strand or the (+) strand and the other strand is identified as the template strand or the (-) strand. The nucleotides in the template strand are the complement of the nucleotides in the coding strand. DNA sequences can originate from either strand, and the sequence data is typically marked with a (+) or (-) sign to identify the strand it belongs to. Both the coding and template strands have its ends identified by the symbols 5′ and 3′, and the 5′ end of the coding strand corresponds to the 3′ end of the template strand, and vice versa. The nucleotides on the complementary strand are typically written from the 5′ to the 3′ end, so it is identified as a reverse complement.

The DNA sequence of an organism is divided into chromosomes; the complete set of chromosomes creates the genome of an organism. Some organisms such as humans are diploid, and have two copies of each chromosome, where each chromosome in a pair is inherited from each of the parents. The size of the genome is quantified as the total number of nucleotides from a strand of a single chromosome from all the pairs of chromosomes in the genome. Organisms are made up of cells, which are the functional units of organisms, and each cell contains a complete genome. The machinery in the cells interpret the instructions in the genome to perform specialised functions necessary to maintain the organism. The number of nucleotides per chromosome and the number of chromosomes vary between different organisms, hence the genome sizes of organisms also vary.

Some parts of the genome consist of sequences of nucleotides that encode for proteins, and these parts of the genome, along with other units of heredity such as ribosomal DNA, are known as genes. The genes and their surrounding areas that give
2.1. GENOMIC DATA

Figure 2.3: Two DNA sequences with mutations.

Various support functionality to the gene are known as coding regions of the genome. The protein-coding genes are translated into proteins to perform their designated function. However, much of a genome consists of non-coding DNA. Although some of these non-coding regions are believed to give support to the coding regions, some of the other functionality is yet to be determined.

An important function of DNA is the translation of protein-coding genes to proteins. Proteins are responsible for performing the biological functions necessary to maintain organisms. For instance, the genes HBA1, HBA2, and HBB create the Hemoglobin protein that transports oxygen around the body in vertebrates. Protein molecules are made up of amino acids, where the sequence of amino acids is determined by the nucleotides in the gene being translated to a protein. The translation occurs by converting triplets of nucleotides (codons) to amino acids. Once the protein molecule is created, it folds into a globular form. The functionality of the protein is dependent on its amino acids and its shape when folded. Determining the shape of a protein molecule from the amino acid sequence is a computationally difficult problem hence, determining the protein function is also difficult.

Although individuals within the same species have the same overall characteristics, there are differences that make individuals unique, and these differences are encoded in the genome as mutations. Genomes of different species also contain commonalities but there may be mutations in these common regions. Therefore, mutations are a key component of DNA sequence collections, and an understanding of the nature of mutations is important for understanding the repeat properties of DNA. We discuss mutations in the next section.

2.1.2 Mutations in DNA

Mutations are differences in a DNA sequence compared to another DNA sequence from the same species or a sequence serving the same functionality. Figure 2.3 illustrates two sequences that are mostly similar except for a mutation of 7 nucleotides at the beginning of the sequences and 3 other single nucleotide mutations in the latter half of both sequences.

Mutations in individual genomes of the same species are common and they represent the distinct characteristics of the individuals, like eye color, hair color and other complex differences. However, certain other mutations can be harmful and could cause various diseases. Mutations can range from single nucleotide changes to insertion and deletion of long segments of nucleotides, and even larger mutations, where
large segments of nucleotides in a chromosome are moved to another chromosome. In this section, we discuss the properties of some of these mutations.

**Single Nucleotide Polymorphisms**

*Single Nucleotide Polymorphisms* or SNPs are single nucleotide differences in individual genomes from the same species, or in chromosome pairs if the genome is diploid. In Figure 2.3, the last three mutations are SNPs, where the nucleotide $g$ at position 20 in the first sequence is replaced by a nucleotide $c$ in the second sequence, resulting in a $g/c$ polymorphism.

SNPs can occur anywhere in the genome. The effect that the SNP has on the organism depends on where in the genome it occurs. If the SNP occurs in a coding region, then in most cases, regardless of which variant of the polymorphism occurs in the sequence, the translated amino acid sequence is the same. This is because multiple codons are mapped to the same amino acid (there are 64 distinct codons and only 20 amino acids, and a stop signal). For example, $tgt$ and $tgc$ both translate to the amino acid *Cysteine*, so a $t/c$ polymorphism on the third position results in the same amino acid. However, if a mutation at the third position changes the triplet to $tgg$ in a coding region, then the *Tryptophan* amino acid is produced. Some such mutations result in individuals having different characteristics, while others can result in a protein that does not function as expected and this could be harmful. If a mutation at the third position changes the triplet to $tga$, then the triplet becomes a signal to stop translating the DNA segment to a protein, hence the translation is terminated prematurely. Generally this is harmful for the organism, since the protein created is either non-functional or may function in a harmful way. As an example, the *BRCA1* gene helps to repair damaged DNA, and certain mutations can lead to the translation of *BRCA1* to end prematurely. The resulting protein is unable to repair DNA, which allows cancerous cells to spread. On the other hand, SNPs that occur in non-functional regions of the genome are less likely to have an effect on the organism, and such mutations could even create new genes that give an advantage to the species.

Apart from single point mutations, other types of mutations such as insertions, deletions, and other large-scale structural rearrangements can also occur in genomes. We discuss these mutations next.

**Insertions and deletions**

*Insertions* and *deletions* are one or more nucleotides that are inserted into or deleted from a DNA sequence, respectively, compared to another *homologous* sequence (sequences that have a common evolutionary origin). Insertions and deletions are commonly identified as *indels* since it is difficult to know whether the first sequence has gained some nucleotides compared to the second sequence or the second sequence
2.1. GENOMIC DATA

Figure 2.4: Two DNA sequences with an insertion and a deletion.

has lost some nucleotides compared the first sequence. An example of an indel in two DNA sequences is illustrated in Figure 2.4. At the 10th position in the first sequence, there is an insertion of length 7 compared to the second sequence, and at position 29, there is a deletion of length 7 compared to the second sequence. It is also possible to interpret the figure as having a deletion of length 7 at the 10th position of the second sequence and an insertion of length 7 at the 29th position, compared to the first sequence.

The length of an indel could vary in size. A common effect of indels in the coding regions of a genome is to cause a frameshift, where the reading frame used to translate codons to amino acids is altered. As an example, the DNA sequence aca cgt tta cgt is translated to the amino acid sequence TRLR. If an insertion of length 1 occurs at the 7th position, resulting in the sequence aca cgt att acg t, then this sequence is translated to the amino acid sequence TRIT. The codons being translated from position 7 onwards are not in the correct reading frame so a different protein is produced. Similar to SNPs, some indels have harmful consequences on the organism, while others remain harmless. Typically, indels in coding regions are in multiples of three due to the detrimental effect of having indels that are not multiples of three. Indels in non-coding regions could have arbitrary lengths.

Large-scale structural rearrangements

The mutations discussed above occur on a small scale within chromosomes and only effect small segments of the genome. Larger scale mutations can occur within chromosomes, where large segments of a chromosome are deleted or duplicated. Such mutations can also occur across chromosomes. Large-scale structural changes are frequent and can occur in human populations at a rate of 1 in 500 births [Hoffee, 2004]. These changes have the potential to cause various diseases, including cancer, where cancer genomes often contain significant rearrangements [Stratton et al., 2009]. In this section we discuss some of these large-scale structural changes.

Figure 2.5 illustrates some large-scale mutations. We first describe intrachromosomal mutations as depicted in the first row of the figure. The first mutation in the row shows the deletion of a large segment of a chromosome. This may result in loss of important genetic material such as genes, which may cause genetic abnormalities. For example, the disease Cri du chat is caused by a missing DNA segment in chromosome 5, while Prader Willi and Angleman syndromes are caused by deletions in chromosome 15 [Hoffee, 2004]. The second mutation shows a DNA segment dupli-
cated within the chromosome. This may result in duplicated genes or chromosomes, which may serve as an advantage, since one of the copies can serve the functions of the organism while the other copies are free to mutate rapidly. This may result in variants of genes that give the organism an advantage. However, duplicate genes and chromosomes can also cause genetic diseases such as Down syndrome, which is caused by some or all of chromosome 21 being duplicated [Hoffee, 2004]. The final mutation in the first row is an inversion. These are mutation events where a segment of the chromosome is detached and then attached back with the ends of the segment reversed. Human chromosome 9 contains such an inversion. As long as nucleotides are not lost or gained in the process, inversions tend not to have an effect on the functioning of an organism [Hoffee, 2004].

The second and third rows of Figure 2.5 show large-scale interchromosomal rearrangements. Insertions are possible, where a large segment of DNA is deleted in one chromosome and that segment is inserted into a different chromosome. Translocations, on the other hand, rearrange large segments of DNA between chromosomes as depicted in Figure 2.5, where a segment of chromosome 4 is swapped with a segment in chromosome 20. For these mutation types, if breaks were to occur within genes, it may lead to genetic defects [Gardner et al., 2011]. For example, chronic myeloid leukemia results from a translocation between chromosomes 9 and 22 [Nowell, 1960]. Many other types of structural variations can also occur, such as isochromosomes, ring chromosomes, and amplification [Gardner et al., 2011; Hoffee, 2004].

The above discussion provides an overview of the types of mutations expected to be observed in DNA sequences, especially when comparing individual genomes from the same species. Mutations can occur due to errors introduced during DNA replication, viruses, chemical reactions, exposure to radiation and many other reasons. The effects of a mutation depends on the region of the genome it occurs in. Generally, mutations to the coding regions can disrupt the organisms normal functioning and lead to certain diseases, hence organisms have the ability to repair mutations occurring in genes. On the other hand, mutations to non-functional regions of a genome may not have any effect, or may lead to a mutation that creates an advantage for the species. Such advantageous mutations are carried through the generations and may result in the creation of new species over time. We briefly discuss evolution in the next section.

2.1.3 Evolutionary conservation

Understanding the role of evolution gives important insights into the level of similarity between genomes of various species, which dictates the compressibility of the dataset. In this section, we give a brief overview of evolution.

Earlier, we described several mutations that occur within genomes. Over time, mutations accumulate in the genomes of species. The changes that promote better
2.1. GENOMIC DATA

Figure 2.5: Several large-scale mutations that can occur in genomes. The first row shows intrachromosomal rearrangements and the last two rows show interchromosomal rearrangements.\footnote{Derived from the image DNA Chemical Structure at en.wikipedia.org/wiki/File:Chromosomes_mutations-en.svg created by the user YassineMrabet, Nov 2009.}

Survival and reproduction are carried through the generations and over time, new species may come into existence. There are observable similarities in the DNA of two species that diverged and evolved separately. In general, the further away in time the divergence occurred, the lower the similarity in the genomes of two species. However, individuals from the same species are likely to have highly similar genomes.

Figure 2.6 illustrates a phylogenetic tree that shows the evolutionary relationship between \( P. \) troglodytes (chimpanzee) and some of the species of the \( Homo \) category (such as \( H. \) sapiens). The amount of similarity between a \( P. \) troglodytes genome and a \( H. \) sapiens genome is likely to be relatively high, given that the two species diverged from their common ancestor around 5-7 million years ago [Chen and Li, 2001]. For instance, the \( H. \) sapiens (human) and \( P. \) troglodytes (chimpanzee) differ by only 6%
CHAPTER 2. DNA BACKGROUND

Figure 2.6: The phylogenetic tree showing the evolutionary relationship between the *P. troglodytes* (chimpanzee) and some of the species of the *Homo* category of species (humans and their closest ancestors).  

```plaintext
gccgctgagtttacctccggctgttcatgc
gccgcctgcct
```

Figure 2.7: A DNA sequence with two exact repeats and one inexact repeat.

in their gene content [Demuth et al., 2006]. The similarity between the genomes of the two species *H. neandarthalensis* and *H. sapiens* is likely to be even higher, given that the two species diverged from their common ancestor around 500,000 years ago [Green et al., 2006]. The similarity levels cannot yet be determined accurately, as the *H. neandarthalensis* genome is still being mapped to the *H. sapiens* genome. The genomes between the different races of *H. sapiens* will be even more similar, since the common ancestors between the different races existed only 200,000 years ago [Stoneking and Soodyall, 1996].

An understanding of the evolutionary relationships between species is important for understanding the level of repetitive segments of DNA that can be observed within and across genomes. Next we discuss types of repeats commonly present in DNA datasets that can be manipulated for their efficient storage.

2.1.4 Repetition in DNA

Repeats in DNA sequences are two or more occurrences of a sequence of nucleotides that are either identical or contain a few variations. Figure 2.7 shows a DNA sequence with two identically repeated substrings and a third inexact repeat containing a single mutation. DNA sequences contain repeats within chromosomes, within a genome, repeats across multiple genomes from the same species or repeats across multiple genomes from different species. The amount of repetition present across a set of sequences is dependent on the amount of similarity between the sequences, which in turn is determined by the evolutionary distance between the species or the function performed by the sequences. Identifying repeats is essential for effective compression of DNA sequence collections.

Repeats can be present in sequences or genomes originating from different species due to the evolutionary relationships that exist between species, as discussed in the previous section. The repeats are most likely to be in the non-coding regions of the

---

2Based on the figure Human Evolutionary Tree at en.wikipedia.org/wiki/File:Human_evolutionary_tree.jpg created by the user adfgf, Jan 2007.
2.1. GENOMIC DATA

Figure 2.8: The similarity between the G. gallus (chicken) and H. sapiens (human) sequences from the hemoglobin HBZ gene. The G. gallus and H. sapiens sequences originates from positions 367–429 and 422–484 of the mRNA sequences (accession: NM_001004374 and NM_005332), respectively.

Korean: cacgccgcctgggacaagttcctatcggtcgtatcctctgtcctgaccgagaagtaccgctga
Chinese: cacgccgcctgggacaagttcctatcggtcgtatcctctgtcctgaccgagaagtaccgctga
Venter: cacgccgcctgggacaagttcctatcggtcgtatcctctgtcctgaccgagaagtaccgctga

Figure 2.9: The similarity between the Korean, Chinese and Craig Venter H. sapiens sequences from the hemoglobin HBZ gene. The Korean and Chinese sequences originates from positions 143974–144399 and the Venter sequence from positions 121656–122081 of chromosome 16.

genomes but they are likely to be approximate due to the mutations accumulated during evolution. Therefore, when comparing DNA sequences, it is natural to allow for mutations. The similarity in sequences originating from the non-coding regions of genomes is likely to be low unless the species being compared have a recent common ancestor, or the non-coding regions perform an important function. Figure 2.8 illustrates the similarity between two small segments of DNA from the hemoglobin HBZ gene. The two sequences are from the G. gallus and H. sapiens genomes, and since diverging from their common ancestor, the two species have evolved separately for around 300 millions years. The two gene sequences are similar except for a small number of mutations, which are underlined. Even with these mutations, the amino acid sequence being produced by these two sequences are almost identical with just two single amino acid differences. This shows that over time, the coding regions of the two genomes have accumulated mutations, but in a way that does not disrupt the amino acid sequence generated.

The repeats occurring in genomes of individuals within the same species are more frequent compared to repeats occurring across genomes of different species. Apart from some mutations, the genomes of individuals are largely similar. The similarity between the genomes of two human individuals is expected to be around 99.5% [Levy et al., 2007]. Therefore, we can expect long exact repeats to be present when comparing such genomes. As an example, the same region of the hemoglobin HBZ gene in the Chinese, Korean and Craig Venter human genomes is depicted in Figure 2.9. The nucleotides in the region are identical.

So far we have discussed repeats that are observed across multiple genomes. However, there are also repeats that occur within individual genomes such as copies of genes and other long segments of DNA. This kind of repetition also captures reverse complement repeats. Figure 2.10 shows a fragment of the H2A gene from
CHAPTER 2. DNA BACKGROUND

Figure 2.10: Segments from four copies of the H2A histone gene in the human genome. The sequences are from positions 1–60 of the H2A gene sequence in chromosome 1 (Accession: NM_003517), positions 1–60 of one of the H2A gene sequences in chromosome 6 (Accession: NM_003510), positions 36–95 of the other H2A gene sequence in chromosome 6 (Accession: NM_170745), and positions 74–133 of the H2A gene sequence in chromosome 11 (Accession: NM_002105).

four separate occurrences in the human genome: one occurrence each in chromosome 1 and 11, and two occurrences in chromosome 6. The gene has two further copies in chromosome 4 and 7, respectively [Strachan and Read, 1999]. The nucleotide sequence contains many mutations but the amino acid sequence encoded from each of these 60 nucleotides is almost identical, with just one amino acid mutation in each of the bottom three sequences, with respect to the first sequence. While these multiple occurrences of the H2A gene are functional, some other genes have copies in the genome that are non-functional due to their lack of protein-coding ability. Such non-functional copies of genes are known as pseudogenes.

Genomes also contain simple repeats, where long sequences of a (poly-a), t (poly-t), di- or tri-nucleotide repeats are commonly found in non-coding regions of the genome. The repeats consisting of two or more nucleotides that are repeated adjacent to each other are known as tandem repeats. For example, the repeated subsequence ttaggg forms a tandem repeat commonly found at the ends of human chromosomes to protect the chromosome ends [Strachan and Read, 1999]. Tandem repeats are one of two main kinds of repeats observed in the human genome. The other is interspersed repeats, which are short (around 130 nucleotides) or long (ranging from around 250 to over 1000 nucleotides) repeats interspersed throughout the genome. These are believed to be inserted into certain parts of the genome to aid new genes to evolve [Strachan and Read, 1999]. Most repeats within a genome are likely to occur in the non-coding regions. But, repeats within genomes are less common than repeats across genomes, especially compared to the repeats in individual genomes from the same species. Around 30% of the human genome is moderately repetitive and only 10% of the genome is highly repetitive [Kass and Batzer, 2001].

Since compression algorithms detect repeats to compress the input, the repeats occurring within genomes and across genomes can be used to compress DNA sequences. Standard compression algorithms that detect exact repeats can be used to compress DNA sequences, but the presence of mutations such as SNPs and indels means that the best possible compression may not be achieved. By detecting exact repeats, a long repeat containing mutations within it will not be identified. Certain
DNA compression algorithms detect approximately similar repeats to improve compression. In Section 3.3, we discuss the issues that arise when compressing DNA sequences and sequence collections, and the approaches taken by various existing DNA compression algorithms. Next we discuss the format in which genomic data, specifically DNA, is stored in digital form.

2.1.5 Genomic datasets

Various techniques are used to determine the DNA sequences of organisms, RNA sequences and amino acid sequences of genes, and the proteins produced by genes and their structural representation. Researchers often need to access and analyse this information and to permit that, the data is typically stored in online databases. Since this thesis focuses on DNA sequences, we only concentrate on how DNA sequences are represented in digital form.

Typically, DNA sequences are stored in text files where nucleotides Adenine, Cytosine, Guanine and Thymine are represented as the ASCII characters a, c, g and t. For human readability, a nucleotide is stored using one byte per nucleotide. Therefore, a text file for a DNA sequence consists of a string of bytes where the bytes are restricted to the ASCII characters a, c, g and t.

However, it is possible that the alphabet for representing nucleotides is larger than just four symbols. When the sequence of nucleotides is determined by experiments, sometimes the identity of certain nucleotides is ambiguous and the ambiguity needs to be represented. For example, if the ambiguity is between a and g then the character r is used to represent the ambiguity, or if the ambiguity is between a, c and g then the character v is used and so on. All the possible combinations of the a, c, g and t nucleotides and their substitutions are as follows:

<table>
<thead>
<tr>
<th>Nucleotide</th>
<th>m</th>
<th>r</th>
<th>w</th>
<th>s</th>
<th>y</th>
<th>k</th>
<th>v</th>
<th>h</th>
<th>d</th>
<th>b</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combination</td>
<td>{a,c}</td>
<td>{a,g}</td>
<td>{a,t}</td>
<td>{c,g}</td>
<td>{c,t}</td>
<td>{g,t}</td>
<td>{a,c,g}</td>
<td>{a,c,t}</td>
<td>{a,g,t}</td>
<td>{c,g,t}</td>
<td>{a,c,g,t}</td>
</tr>
</tbody>
</table>

We identify the nucleotides \{a,c,g,t\} as the standard DNA alphabet \(\sigma\) and the nucleotides \{a,c,g,t,m,r,w,s,y,k,v,h,d,b,n\} as the extended DNA alphabet \(\Sigma\). In practice, most sequences consist of nucleotides from the standard alphabet. From the extended alphabet, the most commonly observed nucleotide is n. In collections such as GenBank, 99% of nucleotides are from alphabet \(\sigma\) and, of the nucleotides from alphabet \(\Sigma\) which are not in \(\sigma\), 98% are the nucleotide n [Williams and Zobel, 1997].

DNA, RNA, and protein sequences are commonly represented in the FASTA format. The format allows for multiple sequences to be in the same file where each sequence is represented as a description followed by the sequence itself, spanning multiple lines. The description begins with the character > and is restricted to a single line. The DNA sequence for the human HBZ gene is shown in Figure 2.11.

20

CHAPTER 2. DNA BACKGROUND

>gi|6633805|ref|NM_005332.2| Homo sapiens hemoglobin, zeta (HBZ), mRNA
accaaggccagttgaggaaggcccaactcagtgcccaccctgcgcgccatgtctc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc
cgacaccatcggcaccgagactctggagaggccatcattgtgctagccattccacgacggc
tctgaccaagacctgagagagaccatcattgtgctagccattccacgacggc

Figure 2.11: The sequence of the human HBZ gene in FASTA format.

Publicly available genetic data is stored in online databases. A key database is Genbank [Benson et al., 2009], which stores the DNA sequences of many organisms and also protein translations of the DNA sequences. Research laboratories submit data from various DNA sequencing projects to the Genbank database. The database contains a large quantity of data. As of 15th October 2010 (Release 180.0), there are 118,551,641,086 nucleotides in the database from 125,764,384 sequences, and the size of the database continues to double every 30 months [Benson et al., 2009]. Another database, Ensembl [Flicek et al., 2011] is a genome database for vertebrates and other eukaryotic species. The database contains complete genome sequences and other detailed information about each genome.

Some other databases contain genomic data from the same or related species. One such resource is PlasmoDB, which contains the DNA sequences of a few different species of Plasmodium responsible for causing malaria. The database also contains gene and protein information, and SNP data, among other information. Similarly, MITOMAP is a specialised database for human mitochondrial DNA. Many specialised genomic databases exist and researchers submit their sequence data to a relevant database. However, it is general practice to also submit the data to GenBank to ensure that at least one genomic database is comprehensive.

The amount of genomic data is growing rapidly, so in this thesis, we focus on algorithms for compactly storing large sets of DNA sequences. Compression algorithms can be used to store DNA sequences in less space than one byte per nucleotide, but at the cost of human readability. We begin our discussion of existing compression algorithms in Chapter 3 to broaden our understanding of the various means in which compression can be achieved, and in Section 3.3 we discuss how some of these compression algorithms are applied to compressing DNA sequences. Next we provide a brief overview of DNA sequencing, and the rapid improvements to the DNA sequencing technologies which is enabling large quantities of DNA to be sequenced.

---

5 plasmodb.org
6 www.mitomap.org
2.2 DNA Sequencing

The technique of determining the sequence of nucleotides that make up a segment of DNA is known as DNA sequencing. The first practical method of DNA sequencing was developed by Frederick Sanger [Sanger et al., 1977]. This technique was time consuming and laborious, requiring several years to determine the sequence of a large genome such as that of the human, even after automation of the process. However, the latest high-throughput sequencers can sequence a large genome in a manner of days. We give a brief description of the methods involved in sequencing and assembly, then discuss large-scale genome sequencing projects, and the nature and quantity of data generated by high-throughput sequencing methods.

2.2.1 Sequencing methods

We begin with a simplified overview of the three main steps of the Sanger sequencing technique [Sanger et al., 1977]. In the first step, the sequence is divided into small fragments and each fragment is cloned several times. The cloned fragments then undergo sequence reactions to allow special nucleotides, containing chemicals to help identify them, to pair with the nucleotides on the fragments. In the final step, the fragments are passed through a medium that detect and record the special nucleotides in the fragments. The fragments are then passed through a sequence assembly algorithm to determine the order in which they are to be arranged to obtain the complete sequence.

High-throughput sequencing has essentially the same principles as Sanger sequencing but with improvements in the techniques used at each step. Each of these techniques also differ between the various sequencing platforms. The main difference compared to Sanger sequencing is that these techniques are more parallelisable, which increases the sequencing speed, thus enabling fast sequencing of large genomes.

Some commonly used platforms are 454, Illumina and SOLiD. Due to the rapid technological advancements, these platforms output much larger quantities of sequence data compared to Sanger sequencing, and at a lower cost. For example, the Illumina HiSeq 2000 sequencer can generate around 150-200 Gbases of 100 base paired end fragments per run, and a run takes approximately 8 days. More than 80% of bases have an accuracy of 99.9%. The machine promises to produce two human genomes in a single run with each nucleotide sequenced approximately 30 times (30× coverage), for $10,000 USD.8 The same sequencer can sequence a chicken genome, which is a third of the size of a human genome, for approximately $1000 USD per genome. In comparison, the ABI 3730xl sequencer that uses the Sanger sequencing method can only output 690–2100 Kbases per day depending on

---

7In paired-end sequencing, the ends of DNA fragments are sequenced and there is a known gap size between the sequenced ends. Paired end reads are easier to assemble.
the chosen sequencing speed\textsuperscript{9} with an accuracy of around 99–99.9\% per base call. Due to the slow speed, at 30\times coverage, 42–130 days are required to sequence a human genome. In terms of the cost, the National Human Genome Research Institute summarised the sequencing costs for their various projects, which states that the cost of obtaining a megabase of sequence data has reduced from approximately $300 in late 2007 to around $0.20 in late 2010.\textsuperscript{10}

The DNA sequence fragments output by sequencing are known as \textit{reads}. Various sequencing technologies produce reads of varying lengths. Sanger sequencers produces reads of around 1000 bases, while 454 and Illumina sequencers produces reads of 400–500 and 35–100 bases, respectively, depending on the sequencer. Sequencers may make erroneous base calls, and the types of errors and error rates vary based on the technology and the sequencer being used. As an example, 454 sequencing tends to produce more indel errors than substitution errors, Illumina sequencers are more likely to produce substitution errors (1-1.5\% error rate on average), and HeliScope sequencers are more likely to produce deletions (2–7\% error rate) than substitution errors (0.01–1\% for a single pass and 0.001\% for two passes) [Shendure and Ji, 2008]. Error rates can also vary within a read.

In the next few years, even more advanced sequencing technologies will be available that avoid the need to clone the fragments, and detect the special nucleotides as soon as they attach themselves to the cloned fragments [Schwartz and Waterman, 2010]. Even more advanced nanopore technology allows a strand of DNA to pass through a tube, where the electrical signals of each base will be measured to determine the nucleotide sequence. The promise of these advanced technologies is to reduce the time taken to sequence a genome, and the cost of sequencing.

Next we provide an overview of sequence assembly, then discuss some of the prominent DNA sequencing projects that are currently underway.

### 2.2.2 Sequence assembly

Sequence assembly is the process of arranging the reads to determine the nucleotide sequence. For each read, an assembly algorithm needs to align both ends of the read to every other read to determine the ordering. Then based on the ordering, the reads need to be positioned to determine the overall sequence. Even with heuristics, sequence assembly remains a computationally expensive step of the sequencing process. We ask the reader to refer to a review by Miller et al. [2010] for further details on assembly algorithms. Sequence assembly is simplified if a reference sequence or a genome exists that closely resembles the sequence being assembled. Then the reads can be aligned to this reference sequence.\textsuperscript{11}

\textsuperscript{10}www.genome.gov/images/content/cost_per_megabase.jpg
\textsuperscript{11}Assembly information, www.ncbi.nlm.nih.gov/genome/assembly/assembly.shtml
Assembled reads produce contigs, which are larger segments of DNA that can then be arranged to infer the complete sequence. If paired-end reads were used, the information can be used to order and orient the contigs to create scaffolds or supercontigs. The contigs within a scaffold may not be placed adjacent to each other, so a series of n nucleotides fill the gap between contigs. For certain projects, only contigs are produced due to the difficulty of constructing full genomes.

The assembly process is affected by read quality and read lengths. Longer reads, such as those from 454 sequencing can be reliably assembled, and therefore long reads are most suitable for de novo sequence assembly, which is assembly without the aid of a reference. However, reads are more expensive to produce. Shorter reads, such as those from Illumina platforms, are harder to assemble, as many reads are likely to overlap with many other reads, making it difficult to determine the correct ordering. The higher error rates, and the inability to distinguish between repeats also makes it difficult to assign a read to a unique place in the assembled sequence. However, shorter reads are faster and cheaper to produce. Also, the quality scores of each read can be used to filter all or parts of the read to reduce assembly errors.

We gave a brief overview of the DNA sequencing process and the performance of high-throughput sequencing machines. Next, we discuss some of the large-scale sequencing projects that use high-throughput sequencing technologies.

2.2.3 Genome sequencing projects

High-throughput sequencing has made large-scale genome sequencing projects practical. Some high-throughput sequencing projects sequence genomes of species for which a sequence does not already exist, while others are resequencing projects which sequence many individual genomes from the same species to better understand the variations present in genomes. As of the 8th of December 2011, there are 3203 sequencing projects registered at ENTREZ Genome Project database, of which only 1153 are complete.\textsuperscript{12} Below we list some of the large-scale sequencing projects.

**1000 Genomes Project**  The 1000 Genomes Project\textsuperscript{13} aims to catalogue the variation present in the human genome to better understand the relationship between the genotype (the genetic information in an organism) and the phenotype (the characteristics and behaviour observed in the organism). At the end of the project, the aim is to have a complete catalogue of SNPs, structural variations and their haplotype contexts (the combination of interacting alleles on a chromosome). The dataset includes approximately 2000 samples that cover 20 populations across the world. The pilot phase of the project undertook low-coverage sequencing of 179 individuals covering 4 populations, high-coverage sequencing of two mother-father-child trios, and exon-targeted sequencing of 697 individuals from 7 populations [The 1000

\textsuperscript{12}www.ncbi.nlm.nih.gov/genomes/static/gpstat.html

\textsuperscript{13}www.1000genomes.org
The results include location, allele frequency and local haplotype structure of 15 million SNPs, 1 million insertions and deletions and 20,000 structural variations, as well as other insightful results about the mutation rates between generations from the trios experiment, and the genetic variation in the neighbourhood of genes from the exon-targeted experiment. The knowledge gathered from this project is invaluable for researchers to find regions of the human genome that are associated with a particular disease or trait by comparing the genomes of people that have the disease or trait to those that do not.

1001 Genomes Project The 1001 Genomes Project\(^{14}\) is of a similar nature to the 1000 Genomes Project, that aims to catalogue the variation in 1001 strains of the plant Arabidopsis thaliana (A. thaliana) to gain a better understanding of the relationship between the genome and the phenotype. The plant is especially interesting, since the availability of naturally inbred strains can provide useful insights into the effect of the environment on organisms, since the genotype of the strains will be very similar.

Genome 10K The Genome 10K\(^{15}\) project aims to sequence the genomes of approximately 16,000 vertebrate species, with approximately one genome per vertebrate genus [Genome 10K Community of Scientists, 2009]. The genomes will represent mammals, birds, amphibians and fish, among others, including some recently extinct species. The project is particularly important for analysing the evolution of vertebrates and answering important questions about the similarities and differences between various types of vertebrates. The project is at its specimen collection stage and will begin once the cost of sequencing becomes affordable.

Sorcerer II Expedition The Sorcerer II Expedition\(^{16}\) is one of the most ambitious sequencing projects underway. The Sorcerer II yacht collects water samples from various seas in order to identify and catalogue the difference in the microbial species in various regions. A test voyage to the Sargasso sea led to the discovery of more than a million new genes, and at least 1,800 species of bacteria. The project provides insights into the microbial organisms that are responsible for maintaining the ecosystem.

The sequencing projects we discussed above are large-scale, and will produce large quantities of DNA data, and provide many insights into the role that DNA plays in the functioning of a wide variety of organisms. Next we discuss the applications in which high-throughput sequencing technologies can be applied, and the opportunities provided by these technologies to advance medical research.

\(^{14}\)www.1001genomes.org
\(^{15}\)www.genome10k.org
\(^{16}\)www.sorcerer2expedition.org
2.2. DNA SEQUENCING

2.2.4 Applications of sequenced DNA

Genome sequencing has a range of applications that allow researchers to better understand the genetic makeup of various organisms, the causes of various genetic and other diseases, and the evolutionary relationship between species. Below we discuss some of the uses of sequenced DNA data.

**De novo sequencing**  De novo sequencing is the sequencing of genomes of species that were previously unsequenced. As the cost of sequencing reduces, projects such as the Genome 10K project [Genome 10K Community of Scientists, 2009] will sequence the genomes of many species that are currently unsequenced. Other possibilities include sequencing genomes of species that are extinct, and some projects that have already been undertaken include sequencing the *H. neanderthal* [Green *et al.*, 2006; Noonan *et al.*, 2006] and the woolly mammoth genomes [Poinar *et al.*, 2006]. Sequencing and analysing these genomes provide valuable insights into the evolutionary history of modern species. One of the most important applications of de novo sequencing is in cancer genomics [Robison, 2010; Stratton *et al.*, 2009]. Genomes in cancerous cells are often highly deranged, hence de novo assembly is necessary to identify the structural changes that have occurred in the genome, which would be hidden if the reads were assembled with respect to a reference genome.

**Resequecing**  Sequencing multiple genomes or a genomic region of interest from multiple individuals within the same species is known as resequencing. Resequecing helps to analyse the variations that exist between individuals. The sequenced genomes are compared to a reference to identify SNPs, indels and rearrangements, among other variations [Miller *et al.*, 2003]. Such an analysis provides insights on how the differences between individuals may be caused by genetic variations. Some examples of resequencing projects are, the 1000 Genomes project for *H. sapiens* and 1001 Genomes project for *A. thaliana* plants, as discussed in Section 2.2.3. Sequencing thousands of entire genomes is a time consuming and expensive process, which has been made feasible through high-throughput sequencing technologies. Targeted resequencing is especially important as many samples of a region of the genome that are associated with a particular disease can be sequenced to analyse variations. Sequencing a small region also makes it cost effective to sequence large sample sets, compared to sequencing entire genomes [Johansson *et al.*, 2011].

**Personal genomes**  Modern medicine, in combination with genome sequencing technologies, can provide personalised medicine based on the genetic sequence of individuals. By sequencing the genome or key parts of the genome of an individual, and identifying SNPs and other variations that may cause certain diseases, early treatment and customised medication can be provided to patients. Currently, the cost of sequencing a complete genome is over $10,000 and the aim is to reduce the
cost to $1,000 in the next few years. Many tests already exist with the capability of analysing human DNA for certain diseases. The Personal Genome Project takes a step in this direction by intending to publish the complete genome and medical information of 1000 volunteers to aid research into personalised medicine.\textsuperscript{17}

**Transcriptome sequencing** The transcriptome of a cell describes the genes actively expressed in a cell under different conditions. Even though each cell contains a complete copy of the genome, cells in different parts of an organism have certain genes active at any given time. An understanding of the transcriptome is necessary for determining the functional elements of a genome and the contents of cells [Wang \textit{et al.}, 2009]. It plays an important role in understanding cancer by analysing cells from infected regions [Maher \textit{et al.}, 2009]. RNA-seq is the use of high-throughput sequencing for transcriptomics [Wang \textit{et al.}, 2009]. Unlike microarrays, this technology allows the analysis of transcripts for organisms whose genomes are not yet sequenced, such as the Glanville fritillary butterfly [Vera \textit{et al.}, 2008]. It is also suitable for detecting variations in the transcribed regions [Cloonan \textit{et al.}, 2008]. Therefore, high-throughput sequencing has revolutionised transcriptomics, making the process more affordable and improving the quality and quantity of data. Wang \textit{et al.} [2009] provides details of the RNA-seq process, its challenges, and applications.

**DNA-Protein interactions** Just as it is important to understand which genes are active in which cells, it is also important to understand the interactions between certain proteins and DNA. The ChIP-seq technology is used to identify the interactions between proteins and DNA to regulate the genes that are expressed. This is important for understanding the gene regulatory network that is associated with various biological processes, which may be vital for understanding disease states [Park, 2009]. In ChIP experiments, the proteins of interest are first bound to the DNA, and once the binding sites are detected, the proteins are unbound. The DNA at the binding sites are fragmented and then sequenced. The sequenced DNA is assembled and mapped to a chromosome or genome to determine the binding sites for the proteins of interest. High-throughput sequencing allows many samples to be sequenced in a single run, which significantly reduces the cost of experiments. The technology also provides significant advantages over microarrays, due to the better resolution, and the ability to cover more of the genome [Park, 2009]. Park [2009] provides further information on the ChIP-seq process, and the advantages and disadvantages of using high-throughput sequencing.

**Metagenomics** Metagenomics is the study of the populations of organisms present in samples obtained from various environments or living organisms [Pignatelli and Moya, 2011]. The samples can originate from water, soil, and blood or stomach

\textsuperscript{17}www.personalgenomes.org
of organisms, among many other environments. The aim is to provide a detailed overview of the composition of various organisms and the functions they perform as a community in that environment. It is also important for analysing the differences in organism compositions across different environments. Prior to high-throughput sequencing, metagenomic studies were restricted to studying a few individual species, and this did not capture the manner in which microbes interact with each other and the environment. The fast sequencing speeds and the low cost of high-throughput sequencing has enabled the metagenomics field to conduct more extensive analysis of samples [Wooley et al., 2010]. Some examples of metagenomics projects involving high-throughput sequencing are the Sorcerer II expedition [Venter et al., 2004] described in Section 2.2.3, and the analysis of microbes in cow rumen to better understand the role microbes play in the degradation of cellulosic plant material [Hess et al., 2011]. Wooley et al. [2010] provides a detailed description of metagenomics, the various issues associated with sequencing, assembly and analysis of metagenomic data, and its applications.

Advancements in DNA sequencing technology have led to the ability to sequence entire genomes at a fast speed and at a low cost, allowing large genome sequencing projects that aim to sequence thousands of complete genomes to be feasible. The quantity of data generated by these projects is extending to petabytes, as sequencing machines produce gigabytes of data per day. This creates interesting challenges for storage, management and analysis of this data. For example, a human genome contains approximately 3 Gbases, and with 30× coverage, the reads for a single genome contains around 100 Gbases. While, the 1000 Genomes Project does not aim to have such high coverage, the data produced by the project for approximately 2000 genomes is still very large. When 2000 human genomes are assembled, the collection will consist of approximately 6 Tbases. Similarly large quantities of data will be generated by many other such projects.

2.3 Chapter Summary

This chapter introduced the basic terminology of DNA, its properties that are of interest to compression, and some large-scale DNA sequencing projects. These projects output large quantities of data, which is the main motivation for the need for efficient algorithms to store, manage and analyse large DNA sequence collections.

After introducing some of the basic terminology associated with DNA, we discussed the various types of mutations such as SNPs, indels and other large-scale rearrangements that can be observed in DNA sequences. Then we introduced the concept of repetitions in DNA, where substrings of DNA can be repeated within and across sequences. Compression algorithms tend to detect repeated substrings in the input and represent these compactly. Therefore, the higher the level of repetition in a dataset, the more compressible it is. We then introduced DNA sequencing,
CHAPTER 2. DNA BACKGROUND

and some of the more prominent large-scale sequencing projects that are currently underway, which were made possible by the recent rapid advancements in the DNA sequencing technologies. Then, a few practical applications of sequenced data were discussed, followed by a brief discussion of the potential sizes of the output from various sequencing projects to motivate the need for efficient storage solutions.

Sequencing centers acquire equipment for storing and managing the large quantities of data generated by sequencing projects. However, research institutes may not have the resources to store, transfer and analyse the data being generated. Therefore, there is a pressing need to improve existing algorithms and data structures for managing these large quantities of data [Wooley et al., 2010]. Many of the larger sequencing projects tend to be resequencing projects, which output collections of highly similar genomes or sequences. Even if the genomes in a collection did not originate from the same species, provided there is some evolutionary relationship between the species, we can expect that there will be some repetitions in the genomes. Therefore, compression algorithms can be used to detect the similarity to store such collections efficiently. It may even be possible to extend the analysis algorithms to use the compressed data instead of the large uncompressed collections to improve the analysis speed and resource use.

In this thesis, we focus on algorithms to efficiently store and access large assembled DNA sequence collections. In particular, we focus on collections of complete genomes, chromosomes and contigs. While efficient storage of read data is also very important, we focus on assembled DNA sequence collections, since in the long run, these are the datasets that researchers will continue to analyse.

In the next chapter, we discuss existing general-purpose and DNA-specific compression algorithms, and evaluate their potential to compress repetitive DNA sequence collections, specifically those generated by genome sequencing projects.
Chapter 3

Compression Background

Large-scale DNA sequencing projects such as the 1000 Genomes project or the Genome 10K project will output large collections of assembled genomes. Collections of this scale are difficult to be stored on standard storage mediums used by researchers analysing this data. Therefore, it may be necessary to compress the data to reduce storage costs. Fortunately, sequences output by projects such as resequencing projects tend to be highly repetitive. Therefore, compression algorithms can be used to store such collections in a compact form to reduce the storage overhead, and compressed collections can even be shared more efficiently.

Simply compressing a large collection may not be sufficient if the sequences in the collection subsequently need to be accessed and searched. Each time a part of the collection is accessed, the entire collection may need to be decompressed, which may be a considerable overhead for large sequence collections. Fortunately, algorithms exist that can compress a collection, then allow fast access to sequences or substrings in the compressed collection, and searching in the compressed collection. In some circumstances, these queries may even be answered faster on the compressed collection compared to answering the equivalent queries in the original collection. This chapter is dedicated to discussing existing compression algorithms, some of which are commonly used general-purpose compression tools, others of which are DNA-specific compression tools, and the remainder of which are tools that compress a collection, and enables access and search queries on the compressed collections.

In Section 3.1, we introduce some basic notation and data structures used throughout the thesis. Then, in Section 3.2, existing general-purpose compression algorithms are discussed. The fundamental ideas of these algorithms are used by tools that compress various types of data such as text, images and videos. Then Section 3.3 introduces the concept of DNA-specific compression and why this type of compression is necessary, before describing some existing DNA compression algorithms. Finally, in Section 3.4, the self-index data structures, and the associated algorithms that compress the input data and provide access to the compressed data are introduced.
3.1 Strings and string data structures

This section introduces the notation and data structures used in the thesis. The notation refers to the string notation and some common queries executed on strings, while the data structures are those used to store strings allowing them to be queried efficiently. More specifically, the data structures described are suffix trees and suffix arrays, and data structures to compactly store arrays of symbols, including bit vectors and arrays of integers, in a searchable form.

3.1.1 String notation

A string is defined as \( T[0\ldots n] \), where \( T \) contains \( n \) symbols and \( T[i] \in \mathcal{A} \) for \( 0 \leq i < n \), where \( \mathcal{A} \) is the alphabet. The empty string is defined as \( \epsilon \). The size of the alphabet is denoted as \( |\mathcal{A}| \). A substring of string \( T \) is \( T[i\ldots j] = T[i]T[i+1]\ldots T[j-2]T[j-1] \), where \( 0 \leq i < n \) and \( i < j \leq n \). The \( i \)th prefix of the string \( T \) is the substring \( T[0\ldots i] \). The \( i \)th suffix of the string \( T \) is the substring \( T[i\ldots n] \). As an example, let \( T = aabcac \). Then \( T[1\ldots 5] = abca \) is a substring of \( T \), \( T[0\ldots 4] = aabc \) is the 4th prefix of \( T \), and \( T[4\ldots 7] = acd \) is the 4th suffix of \( T \).

A lexicographic ordering of strings is defined as follows. A string \( aX \) is lexicographically less than (\(<\)) string \( aY \) if symbols \( a < b \), or if \( a = b \) then \( X < Y \). The empty string \( \epsilon \) is lexicographically less than all other strings that can be generated from the alphabet \( \mathcal{A} \), ie. \( \epsilon < X \), where \( X \in \mathcal{A}^k \) for \( k > 0 \).

Many types of data can be expressed in terms of strings. For example, a DNA sequence is a string \( T[0\ldots n] \) of length \( n \) with each symbol \( T[i] \) obtained from the alphabet \( \{\text{a,c,g,t}\} \). Text documents, HTML documents, protein sequences and source code are just a few other examples of strings.

3.1.2 Queries on strings

Strings may need to be queried to retrieve symbols or information about its contents, and these need to be answered efficiently. The queries listed below form the basis for implementing more complex queries.

- \( \text{display}(s,e) \) returns the substring \( T[s\ldots e] \).
- \( \text{count}(P) \) returns the number of occurrences of a substring \( P \) in string \( T \).
- \( \text{locate}(P) \) returns the positions of occurrences of the substring \( P \) in string \( T \).

For the standard string representation, the \( \text{display}(s,e) \) query is efficient, but \( \text{count}(P) \) and \( \text{locate}(P) \) queries can be expensive for long strings, since a linear search through the string is required to find occurrences of \( P \). Therefore either additional data structures or alternative representations of string \( T \) are necessary to efficiently answer the latter two queries. Other types of queries also exist, such
as querying the existence of a substring and finding the longest common substring. In this thesis, we only focus on the three queries listed above as they are the basic queries on which the complex queries are built.

3.1.3 Suffix trees and suffix arrays

The suffix tree and suffix array data structures described below are used for storing strings so that queries such as count(P) and locate(P) can be answered efficiently.

Suffix trees

String $T$ can be transformed into a suffix tree for efficient pattern matching [Weiner, 1973]. In a suffix tree $ST_T$ of string $T$, the paths from the root to the leaves represent the $n$ suffixes of $T$. The string $T$ is terminated by a symbol $\$$, which is lexicographically less than all other symbols in the alphabet $\mathcal{A}$ to ensure that individual suffixes can be identified when some suffixes are prefixes of other suffixes. The leaves of the tree store the suffix number. Internal nodes of the tree must have at least two children. To enforce this, consecutive edges with one outgoing edge each are concatenated to make a single edge, and the symbols represented by those edges are concatenated and assigned to the new edge. Therefore, each edge represents either a symbol from the alphabet $\mathcal{A}$ or a substring of the text $T$. Since storing the substrings in the edges require $O(n^2)$ space, the edge labels are instead represented as $(i, j)$ for a substring $T[i \ldots j]$. An example suffix tree for the string $T = BANANA\$$ is presented in Figure 3.1. The figure also contains dotted arrows,

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{Suffix_tree_BANANA.png}
\caption{The suffix tree for the text $T = BANANA\$$.
\textsuperscript{1}The dotted arrows are the suffix links.}
\end{figure}

\textsuperscript{1}Derived from the image Suffix tree BANANA at en.wikipedia.org/wiki/File:Suffix_tree_BANANA.png created by the user Nux, Mar 2010.
which gives constant time access to the \((i + 1)\)th suffix from the \(i\)th suffix. These are known as suffix links.

To answer a \(\text{count}(P)\) query, the tree is traversed from the root following edges that represent the symbols in \(P\). If a node is reached where there is no further path to follow, and a suffix of \(P\) still remains unmatched, then the substring does not exist. If there was a path from the root to an internal node that covered the entire substring, then the number of suffixes that can be reached from the internal node are the number of occurrences. Assuming that at each node it is a constant time operation to determine the next node to follow, then the \(\text{count}(P)\) query is an \(O(m)\) time operation, where \(m = |P|\) [Navarro and Mäkinen, 2007]. The \(\text{locate}(P)\) query operates in the same manner as \(\text{count}(P)\), but each of the \(occ\) positions of occurrence \(P\) are also retrieved, hence requires \(O(m + occ)\) time [Navarro and Mäkinen, 2007].

A suffix tree can be constructed in \(O(n)\) time [Farach, 1997; McCreight, 1976; Ukkonen, 1995; Weiner, 1973], but the structure consumes \(\Theta(n \log n)\) bits of space [Kurtz, 1999] with large constants in practice. A more space efficient variation of a suffix tree is the suffix array, which is described next.

### Suffix arrays

The **suffix array** was first introduced by Manber and Myers [1990] as a memory efficient alternative to the suffix tree. A suffix array is a lexicographically ordered list of all the suffixes of a string \(T\) (terminated by \$ so has length \(n + 1\)). The suffix array \(SA_T[0 \ldots n + 1]\) contains a permutation of the integers \(0 \ldots n + 1\), such that \(T[SA_T[0] \ldots n + 1] < T[SA_T[1] \ldots n + 1] < \ldots < T[SA_T[n] \ldots n + 1]\). In other words, \(SA_T[i] = j\) shows that the \(i\)th largest suffix originates at position \(j\) in \(T\). The suffixes are identified using their positions in this manner to use \(O(n \log n)\) bits of space instead of the \(O(n^2)\) space required for storing the suffixes themselves. An example suffix array is illustrated in Figure 3.2.

\[
\begin{array}{ccc}
  i & SA[i] & T[SA[i]..] \\
  0 & 6 & $ \\
  1 & 5 & A$ \\
  2 & 3 & ANA$ \\
  3 & 1 & ANANA$ \\
  4 & 0 & BANANA$ \\
  5 & 4 & NA$ \\
  6 & 2 & NANA$ \\
\end{array}
\]

Figure 3.2: The suffix array for the text \(T = BANANA\$\). The columns are, the indices into the suffix array, starting positions of suffixes in \(T\), and suffixes represented by each \(i\)th entry of the suffix array.
In a suffix array all suffixes with a common prefix are consecutive. For example, all suffixes starting with AN are in $SA_7[2\ldots4]$ in Figure 3.2. Therefore, to determine the positions at which a substring $P$ occurs, two binary searches are required to return the left $l$ and right $r$ boundaries of where $P$ occurs in $SA$. A $count(P)$ query is then evaluated by the expression $r-l$. The $locate(P)$ query can be evaluated by returning all the elements $SA[l\ldots r]$. The $count(P)$ query requires $O(m\log n)$ time, while a $locate(P)$ query requires $O(m\log n + \text{occ})$ time [Navarro and Mäkinen, 2007]. Suffix trees can evaluate $locate(P)$ queries more efficiently than suffix arrays but at the cost of increased space use.

A suffix array representation of string $T$ requires $n\log n + n\log |A|$ bits of storage space for the suffix array and the string $T$. The original construction algorithm proposed by Manber and Myers [1990] requires $O(n\log n)$ time, but newer algorithms exist that can construct suffix arrays in $O(n)$ time [Kärkkäinen and Sanders, 2003; Kim et al., 2005; Ko and Aluru, 2005].

The three basic queries can be answered efficiently using a suffix tree or array representation of a string, but at the cost of increased space usage compared to storing the string itself. Suffix arrays are less powerful compared to suffix trees, since the tree supports many other operations such as access to suffix links and the lowest common ancestor between two nodes using the branching information. Compressed suffix trees exist that internally use compressed suffix arrays [Grossi and Vitter, 2000], but with extra data structures to support tree navigation [Cánovas and Navarro, 2010; Mäkinen et al., 2005; Ohlebusch et al., 2010; Russo et al., 2008; Sadakane, 2007]. Compressed suffix trees and arrays are less efficient at answering the three basic queries compared to the standard representations, but use less space. The representations are only slightly larger than the compressed size of the string $T$ (ie. the $k$th order entropy of $T$, which is discussed in Section 3.2.1).

### 3.1.4 Compact data structures

Compact data structures also exist for arrays of other data types, such as bits and integers. In this section we briefly discuss compressed bit vectors that store bit arrays in a compressed form, and support queries on the compressed array to retrieve information about its contents.

**Compressed bit vectors**

Let $B$ be a bit vector of length $n$ consisting of 0 and 1 bits. Compressed bit vectors support the following two basic queries:

- $rank_b(B, i)$ returns the number of $b$ bits that occur in bit vector $B[0\ldots i + 1]$.
- $select_b(B, j)$ returns the position of the $j$th $b$ bit in bit vector $B$. 
CHAPTER 3. COMPRESSION BACKGROUND

A compressed bit vector stores the bit vector $B$ compactly, while supporting the above two queries. The initial work in the area was by Jacobson [1989], and then later by Munro [1996] and Clark [1996] to represent the compressed bit vector using $n + o(n)$ bits while supporting $\text{rank}_b(B, i)$ and $\text{select}_b(B, j)$ queries in $O(1)$ time. Raman et al. [2002] proposed a solution that requires $nH_0(B) + o(n)$ bits with $\text{rank}_b(B, i)$ and $\text{select}_b(B, j)$ queries requiring $O(1)$ time. The $\text{sdarray}$ representation by Okanohara and Sadakane [2007] uses close to $nH_0(B)$ bits in practice, assuming 1-bits are sparse in $B$. The $\text{select}_b(B, j)$ query operates in constant time but $\text{rank}_b(B, j)$ operates in $O(\log \frac{n}{m})$ time, where $m$ is the number of 1-bits in $B$. We use the $\text{sdarray}$ representation in Chapters 5 and 7.

For representing arrays of arbitrary symbols in a searchable form, the wavelet tree data structure can be used [Grossi et al., 2003]. The array is represented as a tree of bit vectors and the symbols at each position of the array are inferred by traversing the correct path through the tree. This avoids having to store the symbols explicitly, and the bit vectors can be compressed. Rank and select queries can be performed on a wavelet tree. These data structures for compactly representing bit vectors and integer arrays are used in self-indexes, which are discussed in Section 3.4. Next we discuss general-purpose and DNA-specific compression.

3.2 General-purpose compression

We now discuss lossless algorithms for compressing text and other types of data. We discuss the different classes of general-purpose compression algorithms, and their strengths and weaknesses. DNA-specific compression algorithms also apply the same techniques as the general-purpose compression algorithms.

Compression algorithms can be used to store large files more space efficiently than storing the original files. While saving storage space is important, especially when resources are shared among multiple users, compression is also important for efficient data sharing across networks and the internet. Restrictions are imposed by services to limit the amount of data transfer, such as the restrictions that email applications place to limit the size of attachments. Even without such restrictions, it may not be feasible to transfer large quantities of data within a reasonable amount of time using file transfer protocols across a network or even over a USB connection. Therefore, compression algorithms are required to reduce the size of large files to save storage space, and to make data transfer faster. Compression tools have been used widely for this purpose for decades. Some of the frequently used, freely available tools are gzip and bzip2, and some commercial products are WinRAR and WinZip.

Data compression is defined as encoding the data using as few bits as possible. Depending on the purpose of the data, compression can either be lossless or lossy; in the former when data is compressed and subsequently decompressed, no information is lost, whereas in the latter, some information is lost. For images, voice data or
3.2. GENERAL-PURPOSE COMPRESSION

videos, the information lost during decompression is not a concern for most purposes. In this thesis, we focus on lossless text compression algorithms.

In general, compression algorithms follow a three-step process. The first step is to define a set of symbols, which are the individual entities of the compression. For example, in an English text, the set of symbols can be the letters of the alphabet and punctuation, or the distinct set of words in the text.

The second step is to create a model of the input data, where the probability distribution for the set of symbols is determined. In adaptive methods where a symbol is encoded as soon as it is read, the probability distribution is estimated based on the symbols encountered up until that point. In this case, the probability distribution needs to be continuously updated. In non-adaptive methods, the symbol probabilities are determined before encoding begins. The probabilities can also be estimated using similar texts to the ones being compressed. For example, if an English language text is to be compressed, then symbol probabilities can be assigned based on knowledge about the typical probabilities of the letters of the alphabet encountered in these texts.

The third and final step is to encode the symbols by assigning a bit representation to each symbol based on the probability distribution given in the model. The bit representation is identified as a codeword. Typically, shorter bit sequences are assigned for more frequent symbols and longer bit sequences are assigned for less frequent symbols. When decoding, the decoder has access to the same probability distributions as the encoder at each step [Witten et al., 1999]. Several encoding techniques assign bit representations to encode the symbols based on the probability distribution. During compression, integers and other symbols that are not part of the original text need to be encoded to represent meta-data that assists during decompression. Special encoding techniques are used to encode these.

The class of algorithms that replace substrings of symbols with a single symbol that refers to an entry in a dictionary of substrings are known as dictionary compression algorithms. The class of algorithms that encode a single symbol at a time based on the probability distribution of each symbol are known as statistical compression algorithms [Witten et al., 1999]. Statistical compression algorithms rely on having an accurate probability distribution of the symbols to encode each symbol efficiently. The performance of dictionary compression algorithms on the other hand rely on the substrings of symbols that are chosen to be replaced by a dictionary entry. In Section 3.2.2 we describe dictionary compression and in Section 3.2.3 we describe statistical compression, respectively, including examples of algorithms that belong to each of these classes. In Section 3.2.4 we also describe a technique known as the Burrows-Wheeler Transform that can transform data into a form that is more compressible. In Section 3.2.5 we describe some encoding techniques that are used in the last step of compression. Prior to describing the algorithms, we discuss performance measures that can be used to analyse the compression performance of algorithms.
3.2.1 Compression performance

To compare compression tools, it is necessary to define a set of criteria that can be used to compare the performance consistently. In this thesis, we use the following three measures: compressed size, compression and decompression speed, and the memory used during compression and decompression. Compression algorithms make trade-offs between these three properties. In general, “the better the compression, the slower the program runs or the more memory it uses.” [Witten et al., 1999]

Compressed size

Compressed size is typically measured in terms of the average number of bits used to encode each symbol in the input. The lower the average number of bits per symbol, the better the compression. However, for any given input sequence, there exists a lower bound on the average number of bits required to encode each symbol, which is known as the information content, or formally as the entropy of the sequence [Shannon, 1948]. It is defined as follows:

\[
H_0(T) = -\sum_{i=1}^{h} \frac{n_i}{n} \log \frac{n_i}{n}
\]  

(3.1)

where \(T\) is the text to be compressed, \(n = |T|\) is the number of symbols in the text, \(n_i\) is the count of the \(i\)th symbol in the alphabet and \(h\) is the number of distinct symbols. It is assumed that all symbols in the input alphabet occur at least once. We used the symbol \(H_0\), since Equation 3.1 assumes each symbol occurs independently of other symbols. We discuss \(k\)th order entropy shortly, where the context of a symbol is taken into account.

Given that \(H_0\) is the average number of bits per symbol, \(nH_0(T)\) is the compressed size of input \(T\) if an ideal context-free compressor is used. Intuitively, if a symbol has a high probability, then it is more likely to occur than not, so it conveys little information, and a small amount of bits need to be transmitted to represent it. If the probability of a symbol occurring is low, then the information represented by an occurrence of this symbol is high, so the number of bits required to transmit this symbol is higher. The information content is maximised when the probability distribution for all the symbols is uniform, therefore all symbols receive a unique codeword of the same length. Methods such as Huffman coding and arithmetic coding can encode an input close to the lower bound specified by the entropy.

Equation 3.1 assumes that the probability of a symbol is independent of the probability of any other symbol. However, if the last \(k\) symbols that occurred before the current symbol are considered, then more information is available on what symbols tend to occur after certain other symbols. Therefore, a more accurate probability distribution of each symbol can be determined by considering the context. The \(k\)th order entropy is the lower bound on the output size of an ideal compressor that
3.2. GENERAL-PURPOSE COMPRESSION

encodes each symbol in $T$ based on the probability of the symbol given $k$ symbols that appeared before it. It is defined as follows:

$$H_k(T) = -\frac{1}{n} \sum_{w \in \sigma^k} \left( n_w \sum_{i=1}^{h} \frac{n_{w\alpha_i}}{n_w} \log \frac{n_{w\alpha_i}}{n_w} \right)$$ (3.2)

where $\sigma$ is the alphabet of symbols in $T$, $\sigma^k$ is the set of all substrings of length $k$ from alphabet $\sigma$, $n_w$ is the number of occurrences of the substring $w$ in text $T$, and $n_{w\alpha_i}$ is the number of occurrences of the substring $w\alpha_i$ in text $T$, where $w$ is a substring of length $k$ from $\sigma^k$, and $\alpha_i$ is a symbol from $\sigma$. The higher the $k$, the lower the $k$th order entropy. In other words, $H_0 \geq H_1 \geq H_2 \geq \ldots \geq H_k$ for any $k$. Compression tools aim to achieve a compressed size measured in bits per symbol that is close to the $k$th order entropy.

To obtain the average number of bits used by a tool to represent a symbol, the final compressed size in bits is divided by the number of symbols in the original input. This value can be used to directly compare the compression performance of various tools. Those that have the lowest number of bits per symbol (bps) have produced a better compression result than those with a higher number of bits per symbol. The bits per symbol value of a tool can also be compared against the $H_k$ value of the dataset to analyse the performance of a tool. In practice, $H_k$ is an unknown quantity for an arbitrarily large $k$. In this thesis, we compare the results of compression tools to the 0-order entropy $H_0$ of a dataset. For DNA sequences, the preferred terminology is to measure the compressed size in bits per base (bpb) rather than bits per symbol, since nucleotides are often referred to as bases.

**Compression and decompression speed**

Compression and decompression speeds are measured by the number of seconds taken to compress the input data and to decompress the compressed data. The speed can also be represented as the number of symbols compressed or decompressed per second. While faster compression is favourable, the speed may be achieved at a cost to the compressed size. Once a dataset is compressed, it is likely to be decompressed frequently to access parts of the compressed data, but it will only need to be re-compressed if the data is modified. Therefore fast decompression speeds are more important than fast compression speeds.

**Memory usage**

In this thesis, memory use is measured as the maximum number of bytes of memory occupied by a tool during compression or decompression. Typically, memory use is dependent on the size of the dataset being compressed. Tools such as gzip compress using memory that is independent of the input size but this efficiency is achieved at a cost to the compressed size.
CHAPTER 3. COMPRESSION BACKGROUND

In Chapter 4, we compare various compression tools in terms of the three measures described above. In chapters 5 and 6, the three measures are used to evaluate the implementations of the two compression algorithms introduced in this thesis. Next we discuss general-purpose compression algorithms.

3.2.2 Dictionary compression

We begin our discussion of general-purpose compression algorithms with the concept of dictionary compression. Algorithms of this type detect repeats to compress the input, making this type of compression ideal for compressing DNA sequences. Therefore, most DNA compression algorithms are based on dictionary compression, including the two algorithms we present in this thesis.

Dictionary compression algorithms compress the input by replacing repeated substrings with symbols, known as non-terminals (terminals are the symbols in the input alphabet) that reference the entries of these substrings in a dictionary. What constitutes a substring is determined at the symbol definition step of the compression process, and these could be fixed or variable length substrings. The dictionary stores substrings that are repeated in the input text. Compression is achieved if the codewords representing the non-terminals have a shorter bit representation than the encoding for the original substring.

Dictionary compression algorithms differ in the way in which the dictionary of repeated substrings is constructed. We discuss three possibilities for constructing a dictionary. One is to use an existing dictionary that is suitable for the data being compressed. For example, a dictionary containing common English words such as ‘and’ and ‘the’ can be used to compress English texts. The compression performance of using a static dictionary will be good as long as the repeats in the dictionary represent the repeats in the data being compressed. But in general, a static dictionary will not capture the particular repeat properties of a dataset, and better compression can be achieved by using a dictionary that is generated specifically for the dataset being compressed.

Another approach is to use an initial pass over the input data to find repeated substrings to build a dictionary. The dictionary can then be used to encode the input in another pass through the data. The other approach is to use a single pass to compress the input by dynamically maintaining a dictionary based on the input seen so far, and using this dictionary to encode repeated substrings as they are encountered. The algorithms that are based on the latter two dictionary construction methods are identified as semi-static and adaptive dictionary compression algorithms, respectively.

In semi-static dictionary compression, the dictionary construction and substitution steps can be repeated multiple times until the desired compression result is achieved. The dictionary must be part of the compressed data to allow the de-
compressor to decode the non-terminals, and this can be an expensive overhead. If the substrings that belong to the dictionary are chosen well and the non-terminals are assigned appropriate codewords, then semi-static dictionary compression algorithms can produce good results. This type of compression has two main drawbacks. Firstly, the dictionary may have a large memory overhead if the dataset being compressed has many distinct repeats or if care is not taken in choosing an appropriate set of substrings for the dictionary. Secondly, multiple passes over the input can be expensive, especially when the input is large.

Adaptive dictionary compression methods on the other hand compress the input in a single pass by maintaining a dictionary of the previously seen input. When a substring that occurred earlier in the input is encountered, it is encoded as a reference to the earlier occurrence. An explicit dictionary is not required since at any point during decompression, the decompressor has access to the same set of already seen symbols as the compressor did when compressing at that point of the input. Ideally, all of the so far seen input should be available to the compressor so that repeated substrings that are far apart in the input can be detected and encoded. However, if the input text is large, this is infeasible and only a limited amount of the previously seen input can be kept in memory at any given time. As a consequence, repeated substrings that are far apart in the text will not be detected, hence compression achieved by these algorithms may be worse than that of semi-static algorithms.

Even with these limitations, adaptive algorithms are commonly used as a result of the faster compression speed and low memory overhead of the dictionary. Most adaptive algorithms are based on the Lempel-Ziv method. The two main Lempel-Ziv approaches are discussed next. Semi-static dictionary compression is discussed later in this section.

Adaptive dictionary compression

Below we describe the two adaptive dictionary compression techniques, LZ77 and LZ78. We give particular attention to LZ77, since it is referred to throughout the thesis, and our RLZ algorithm is based on the principles of LZ77. We also discuss another adaptive algorithm known as Sequitur.

LZ77 compression The LZ77 [Ziv and Lempel, 1977] algorithm is a commonly used compression techniques. The algorithm substitutes a repeated substring as a reference to an earlier occurrence of the substring in the input. The cost of encoding the reference is less than encoding the substring itself, especially if the repeat is long. When decompressing, the data to the left of the current position have already been decoded. So when the next reference is read, the decompressor has access to exactly the same data that the compressor had access to when creating that reference, allowing these algorithms to decompress without using an explicit dictionary.

The LZ77 compression algorithm is presented in Algorithm 1. For an input text
Algorithm 1  

**LZ77**\_compress\((M, n, W)\) takes a string \(M\), the string length \(n\), and sliding window length \(W\) as input, and returns the compressed string, where repeated substrings are replaced by references to earlier occurrences of the substrings.

1: \(i \leftarrow 0\)
2: **while** \(i < n\) **do**
3: \(\text{found} \leftarrow \text{search for longest prefix of } M[i \ldots n] \text{ in } M[i - W \ldots i]\)
4: **if** \(\text{found}\) **then**
5: \(m \leftarrow \text{starting position of match}\)
6: \(c \leftarrow i - m, l \leftarrow \text{length of the match}\)
7: **encode** \(\langle c, l \rangle\) \{encode the position offset and length pair\}
8: **else**
9: \(l \leftarrow 1\)
10: **encode** \(\langle M[i] \rangle\) \{encode a single literal\}
11: **end if**
12: \(i \leftarrow i + l\) \{advance to the next position to start compressing from\}
13: **end while**

\(M\), if \(i\) symbols have been encoded already, then the substring \(M[i \ldots i+l]\) is encoded by a position offset and length pair \(\langle c, l \rangle\), where \(l \geq 1\), if this substring occurred previously at an offset \(c\) from position \(i\) (lines 4–7). If there is no such substring, then symbol \(M[i]\) must not have been encountered before, so it is encoded as a single literal \(\langle M[i] \rangle\) (lines 8–10). The output of the encoder is either position offset and length pairs or single literals, and we identify these as *factors* for the remainder of the thesis. In Section 3.2.5, we discuss some techniques that can be used to encode the position offsets, lengths and single literals. The **LZ77** decompression algorithm is presented in Algorithm 2. When decoding, single literals are decoded as is (lines 6–8), while position offset and length pairs \(\langle c, l \rangle\) are decoded by looking back by \(c\) positions on the already decoded text and copying \(l\) symbols (lines 3–5).

To determine if the next substring to be encoded has occurred previously or not, the already encountered symbols need to be stored in an efficiently searchable representation such as a hash table [Witten *et al.*, 1999], a suffix tree or other specialised data structures [Larsson, 1999]. However, it is impractical to store all of the previously seen input in main memory and to search it, especially for large inputs. Therefore, the input stored in memory is usually restricted to a sliding window of the last \(W\) symbols encountered, which is continuously updated as the input is encoded. In general, the shorter the sliding window size, the lesser the amount of repeated substrings that can be detected, and higher the compressed size. For inputs such as text documents, a small window size is sufficient, since the repeated words tend to be closer together. The popularity of **LZ77** implementations arise from the fact that they are able to compress text well using a small amount of memory and at a fast speed. However, the experimental results in Chapter 4 show that the compression performance of certain **LZ77** implementations are not so promising, due to the repetitions in DNA being far apart.
3.2. GENERAL-PURPOSE COMPRESSION

Algorithm 2 \texttt{LZ77.decompress}(M', n') takes compressed text \( M' \), and number of factors in the compressed text \( n' \) as inputs, and returns the decompressed text \( M \).

\begin{algorithm}
\begin{algorithmic}[1]
\State \( i \leftarrow 0, j \leftarrow 0 \{ i \) is the number of symbols decoded, \( j \) is an index into \( M' \}\}
\While{\( j < n' \)}
\If{\( M'[j] \) is \( \langle c, l \rangle \)} \Comment{factor is a reference}
\State \( M[i \ldots i + l] \leftarrow M[i - c \ldots i - c + l] \) \Comment{copy the earlier occurrence}
\State \( i \leftarrow i + l \)
\Else \Comment{factor is a single literal}
\State \( M[i] \leftarrow \) \text{decoded single literal} \( M'[j] \)
\State \( i \leftarrow i + 1 \)
\EndIf
\State \( j \leftarrow j + 1 \)
\EndWhile
\State \textbf{return} \( M \)
\end{algorithmic}
\end{algorithm}

A frequently used, fast and efficient LZ77 implementation is \texttt{gzip} [Deutsch, 1996]. The implementation keeps the substrings that occurred in the last \( W \) sized window in a hash table, where the first three symbols of a substring are used to calculate the hash. Each hash table entry is a linked list of substrings that begin with the three symbols. When the longest matching substring needs to be found, the first three symbols are used to find the relevant entry in the hash table. If it exists, the corresponding linked list is traversed until the longest possible match is found.

The window size \( W \) is specified indirectly through the compression speed. The slower the compression speed, the larger the window size and higher the compression achieved. Therefore better compression comes at the cost of higher memory use and slower compression speed. However, there is a maximum limit of 32 Kbytes for the window size [Deutsch, 1996], so for files where the repeated substrings are further than 32 Kbytes apart, \texttt{gzip} will not achieve good compression performance.

The \texttt{7-Zip} file archiving tool implements variants of the LZ77 algorithm such as \texttt{gzip} and \texttt{zip}, but \texttt{LZMA} is the most effective LZ77 implementation available in the tool. The tool permits a maximum 1 Gbyte window size in conjunction with the \texttt{LZMA} implementation. Therefore, unlike \texttt{gzip}, \texttt{7-Zip} can detect repeats that are much further apart and can achieve much better compression as a result. Given the large window size, \texttt{7-Zip} uses the following method to efficiently find the longest matches with its hash table [Salomon \textit{et al.}, 2009]. In the fast variant, each hash table entry contains a list of locations of substrings in the sliding window beginning with a distinct \( k \)-mer, where \( k \) is limited to 2, 3 or 4 bytes. To find the longest match, only a user-defined number of entries in a list for a given \( k \)-mer are checked. In the normal variant, each hash table entry contains a pointer to a binary search tree with nodes containing locations of substrings in the sliding window beginning with a distinct \( k \)-mer. To find longest matches, the tree corresponding to the \( k \)-mer of interest is used to access the positions in the sliding window to conduct the matching.
CHAPTER 3. COMPRESSION BACKGROUND

The compression and decompression speeds of 7-Zip are slower than that of gzip, but the gain in compression for large inputs outweighs the slow speed. As an archiving tool, 7-Zip can also compress multiple files into a single archive, which also allows the decompression of individual files in the archive. This can also be achieved with gzip by first creating a tar file. However, decompressing files appearing later in the archive requires access to files earlier in the archive, hence the speed of decompression depends on the position of the file in the archive. We compare the performance of 7-Zip and gzip in Chapter 4 for compressing DNA sequences. Next we discuss another adaptive dictionary compression method known as LZ78, that uses an alternative technique for storing and searching the symbols within the sliding window.

**LZ78 compression** Like LZ77, LZ78 algorithms also replace repeated substrings with references to an earlier occurrence [Ziv and Lempel, 1978]. The main difference is that LZ78 keeps a dictionary of substrings with unique non-terminals assigned to each entry, and these non-terminals are used to substitute repeated substrings instead of a position offset. Like LZ77, the dictionary is not explicitly stored and is inferred during decompression.

Formally, assuming $i$ symbols of an input text $M$ are already encoded, and $M[0...i]$ is stored in a dictionary $D$, then the longest matching prefix of the remaining input $M[i...n]$ is searched for in $D$. If the longest match has a length $l \geq 1$, the non-terminal $j$ for the matching substring (obtained from $D$), and the next symbol in the input $x = M[i + l]$ are encoded as $\langle j, x \rangle$. The substring $M[i...i + l + 1]$ is then added to the dictionary with a new non-terminal. If there was no match, then symbol $M[i]$ is encoded as a single literal $\langle 0, M[i] \rangle$, where 0 indicates this factor is not a reference to an earlier occurrence. Symbol $M[i]$ is also added to the dictionary. Initially, the dictionary is empty so individual symbols are encoded as single literals. Later in the compression, most substrings are likely to have been encountered before, so they are encoded as references. A popular variant of the LZ78 algorithm is LZW, whose factors do not contain the next symbol in the input, and there are no single literal factors, since the dictionary is initialised with symbols appearing in the input. The widely used implementation of LZW is the Unix utility compress.

The LZ78 algorithm creates a new dictionary entry as each new symbol is read from the input text. Since each new substring of length at least two added to the dictionary is one symbol longer than an existing entry in the dictionary, each substring in the dictionary is unique. This constitutes the main difference between LZ77 and LZ78, where the longest matching substring in LZ78, if it exists, is unique unlike the many potential matches that can be found in an LZ77 dictionary. The LZ78 dictionary can be represented as a trie, so searching for the longest matching substring is more efficient in LZ78 compared to LZ77. Unfortunately decompression is slower since the trie needs to be constructed when decoding [Witten et al., 1999].
Even with the trie representation, it is expensive to store all symbols from the input seen so far. When a certain size limit for the dictionary is reached, it is possible to stop updating the dictionary, or the dictionary can be reinitialised, or it can be rebuilt with the last few hundred bytes of the already encoded text [Witten et al., 1999]. With these restrictions, all of the previous input will not be visible to the compression tool, so repeats that occur certain distances apart will not be detected.

Another drawback of LZ78 is that substrings are added to the dictionary arbitrarily. Therefore, some entries may never be referred to, so the dictionary consumes more space than necessary. Also, for entries that are only referred to once, it may be more efficient to encode the substring itself. Next we discuss the Sequitur algorithm which is more selective on the substrings being added to the dictionary.

The Sequitur algorithm Sequitur [Nevill-Manning and Witten, 1997b,c] generates a context-free grammar of the input, then the grammar is encoded to produce the compressed output. Like LZ77 and LZ78, the input is compressed in a single pass. But unlike the dictionaries generated by LZ77 and LZ78, the context-free grammar can be used to identify interesting hierarchical structures in the input stream.

A context-free grammar (CFG) is a set of rules that generate a formal language. A rule is represented as $A \leftarrow B$, where $A$ is a non-terminal, and $B$ is a combination of terminals and non-terminals. Terminals are symbols from the input alphabet, and non-terminals are identifiers representing substrings of terminals and other non-terminals, thus allowing the grammar to be hierarchical. A string in the language is derived beginning with the top-level rule $S$ and recursively substituting the non-terminals on the right-hand side of $S$. A context-free grammar has no rules where two rules generate the same string.

Sequitur begins with an empty non-terminal $S$. A symbol at a time from the input is appended to $S$. If the new symbol results in there being two copies of the same substring in $S$, then both occurrences are replaced with a new non-terminal, and the rule describing the substitution is added to the grammar. If the new symbol results in a substring in $S$ that already has a rule, that substring is replaced by the corresponding non-terminal. Once all the symbols are read, the input text is compactly represented by $S$ as a series of terminals and non-terminals. The remaining rules represent the redundancy. The grammar has two properties; diagram uniqueness, where no pair of adjacent symbols appear more than once in the grammar, and rule utility, where every rule is used more than once. The first rule ensures that every rule is unique and the second ensures that a rule is useful.

The algorithm operates in linear time. Due to the hierarchical nature of the replacements, it is possible to have rules in the grammar that are only used once at the end of compression. Sequitur replaces such non-terminals with the substrings they represent, thus eliminating rules from the grammar that are not used at least twice. Only rule $S$ is encoded, and a minimum amount of information to help the
decompressor construct the remaining rules is included in the compressed output.

The compression results presented by Nevill-Manning and Witten [1997a] show that Sequitur achieves better compressed sizes than gzip and compress. This is expected, since Sequitur detects repeats globally. However, the speed and memory usage of the algorithm is also likely to be much higher than a LZ77 or LZ78 implementation. It is also unlikely that Sequitur would produce better results than a semi-static dictionary compression algorithm since only a single pass through the input is used to greedily make substring replacement decisions. To achieve good compression, potential repeats should be detected in advance so that overlapping repeats can be analysed to make better substring replacement decisions. Next we discuss semi-static dictionary compression algorithms that pay more careful attention to the substitutions that are made, using Re-pair and RAY algorithms as examples.

Semi-static dictionary compression

As discussed in the previous section, adaptive dictionary compression algorithms are fast and memory efficient but the efficiency comes at a cost to the compressed size. Semi-static dictionary compression algorithms on the other hand use more time and memory to build the dictionary of repeats. The quality of the dictionary built depends on the size of the input, the memory use limits and the required compression speed. Below we describe two semi-static dictionary compression algorithms, Re-pair and RAY, that use slightly different approaches to constructing a dictionary. The discussion focuses on RAY, since our COMRAD algorithm is based on its principles.

The Re-pair algorithm  Re-pair [Larsson and Moffat, 1999] uses a simple technique to ensure that the most frequent repeats are substituted by non-terminals before less frequent repeats. Initially, Re-pair reads the entire input and calculates the frequencies of all adjacent pairs of symbols. Then the most frequent symbol pair ab is chosen and substituted by a non-terminal A, and the rule $A \leftarrow ab$ is added to a dictionary. The frequency counts are updated to reflect the input after the substitutions. Then the next most frequent symbol pair is chosen and its occurrences are substituted. The process continues until all symbol pairs occur only once. Like Sequitur, the dictionary constructed by Re-pair is hierarchical, since substituted symbol pairs can contain non-terminals. Finally, the compressed text and the dictionary are encoded to obtain the final compressed output.

During decompression, first the dictionary is decoded, then the text is decompressed using the dictionary. There are two options for storing the dictionary in memory. One is to store the dictionary of substitution rules. When a non-terminal is decoded, it may be necessary to recursively descend into the non-terminals on the right-hand side of the rule. Recursive expansion has to be conducted as many times as there are occurrences of the non-terminal, making decompression slow. Instead, the option adopted by Re-pair is to store the dictionary with the non-terminals ex-
3.2. GENERAL-PURPOSE COMPRESSION

Panded. This is less memory efficient than the former approach but decompression is faster, since the same non-terminal is not decompressed several times.

Larsson and Moffat [1999] describe an efficient implementation that does not require multiple passes through the input, and allows the input to change dynamically as symbol pairs are substituted. For an input of length \( n \), Re-pair compresses in \( O(n) \) time, using at most \( 5n + 4\sigma^2 + 4d + \lceil \sqrt{n} \rceil \) words of memory, where \( \sigma \) is the number of symbols in the alphabet of the input sequence \( M \), and \( d \) is the total number entries in the final dictionary. Navarro and Russo [2008] showed that the size of the compressed output is at most \( 2nH_k(M) + o(n \log \sigma) \) bits for any \( k = o(\log n) \).

The experimental results by Larsson and Moffat [1999] show that Re-pair compresses better than gzip, which is unsurprising since, unlike gzip, Re-pair has access to a comprehensive dictionary of repeats present in the input. However, Re-pair takes longer to compress and decompress than adaptive algorithms, and will also require more memory to store the input and frequency counts.

Both Re-pair and Sequitur are similar in the sense that each non-terminal substitutes a repeated symbol pair, and both algorithms detect repeats globally. There has been no comparison of the performance of the two algorithms that we are aware of. However, we expect Re-pair to produce better compressed sizes, since the set of substitutions are chosen carefully based on the frequency of repeats, unlike the greedy strategy adopted by Sequitur. A drawback of Re-pair is that the input is stored in memory, which limits the size of the datasets that can be compressed. Nevertheless, Re-pair shows that semi-static dictionary compression is efficient and can produce good results. In Chapter 8, we explore the potential of a Re-pair or Sequitur generated dictionary to be used as a reference string for our RLZ algorithm.

The RAY algorithm RAY [Cannane and Williams, 2001] is similar to Re-pair, except, the occurrences of many distinct symbol pairs may be substituted during an iteration. The algorithm also supports random access into the compressed data. Unlike Re-pair and Sequitur, RAY is a multi-pass algorithm.

A string \( M \) and a frequency threshold \( f \) are the inputs for RAY. First, the frequency counts of symbol pairs are recorded. Each iteration of the algorithm consists of three steps. In the first step, the symbol pairs with a frequency above the threshold \( f \) (candidates) are identified. Like Re-pair, RAY favours the substitution of more frequent symbol pairs over less frequent ones. Each candidate has a candidate count that measures the number of occurrences of that symbol pair that will be substituted. Candidates are identified by comparing symbol pairs in all triplets of symbols in \( M \). If the left-most symbol pair of a triplet has a equal or higher count than the right-most symbol pair, then it is chosen as a candidate and its count is updated. If the left-most symbol pair has a lower count than the right-most pair, then the triplet window is advanced until a triplet is encountered where the left-most symbol pair is more frequent than the right-most pair. This is necessary to ensure that the
globally best symbol pairs are chosen as candidates.

At the end of the first step, only non-overlapping symbol pairs have a candidate count greater than zero, and these symbol pairs are chosen for substitution. In the second step, each distinct symbol pair is assigned a non-terminal and added to a dictionary, and the substitutions are made in the input. In the third step, the symbol pair frequencies are updated to reflect the input after the substitutions are made. The three steps are executed for many iterations until the input stream does not contain any symbol pairs with a frequency of at least \( f \). Finally, the dictionary of substitution rules and the modified input text are encoded for storage.

The decompression algorithm of RAY is similar to Re-pair. The dictionary can either be stored in memory as the hierarchy of rules, or the right-hand sides of rules can be expanded to support fast decompression at the cost of storing expanded substrings. As each symbol is decoded, if it is a non-terminal, the dictionary is used to retrieve the substring represented by that non-terminal.

Compression results of RAY presented by Cannane and Williams [2001] showed that, overall, RAY compresses better than gzip and compress, has faster decompression, but slower compression. Random access speeds were also analysed and showed that RAY can access substrings from the compressed dataset faster than accessing the same segments from the uncompressed collection on disk.

RAY also has a few drawbacks. The algorithm requires multiple passes through the input to find repetitions. Also, since repeats are detected globally, the memory usage of the dictionary may be high for large inputs. Some of these issues are resolved by the XRAY algorithm [Cannane and Williams, 2002], which first compresses a sample of the input to determine the dictionary, and codewords for terminals and non-terminals. Then the entire input is compressed in a single pass using the dictionary and codewords from the sample compression. Even though the memory consumption of semi-static dictionary compression algorithms is reduced by this technique, choosing an appropriate sample is non-trivial.

Although RAY and Re-pair are similar, RAY would require less iterations to compress than Re-pair, since more than one distinct pair of symbols can be substituted in an iteration. The two algorithms are likely to produce similar compressed sizes, since substitution decisions are made according to the frequencies of symbol pairs. Since RAY detects global repeats and uses less iterations, we base our COMRAD algorithm on the principles of RAY. COMRAD is presented in Chapter 5.

Summary
In this section, we discussed two main types of dictionary compression algorithms, namely adaptive and semi-static compression. Which type of algorithm is suitable for a particular application depends on which aspects of compression performance are more important. If low memory use and fast compression speeds are important,
3.2. GENERAL-PURPOSE COMPRESSION

then an adaptive algorithm should be used. However, if the compressed size is more important, then a semi-static algorithm is more suitable.

Determining which dictionary compression method gives the best dictionary of repeats is difficult. The best dictionary can be defined in terms of the *smallest grammar problem*, where an input string is represented by a formal grammar and the size of the grammar is minimised. The size of the grammar is defined as the number of symbols on the right-hand side of the substitution rules. Charikar *et al.* [2005], based on the work of Storer and Szymanski [1982], showed that for the smallest grammar problem, approximating the size of the smallest grammar to within a small constant factor is *NP-hard*. Algorithms such as *LZ78*, *Sequitur*, *Re-pair* and *RAY* are heuristics that are approximations to the smallest grammar problem.

3.2.3 Statistical compression

Dictionary compression methods discussed in the previous section substitute repeated substrings in the input with shorter codewords to compress the input. The compression effectiveness is dependent on the substrings chosen to be substituted by non-terminals. Statistical compression on the other hand uses the probability distribution of symbols to compress the input, where the distribution can be determined by a semi-static or an adaptive approach. Here we focus on the adaptive approach.

In an adaptive model of statistical compression, the algorithm begins with an initialised probability distribution, which is updated as each symbol is read. As an example, for an English text, the initial probability distribution has equal probabilities of $\frac{1}{128}$ for the ASCII characters. If character *a* is read from the input stream, it is encoded based on its current $\frac{1}{128}$ probability. Then the probability of *a* is updated to $\frac{2}{129}$, while the probabilities of other characters are updated to $\frac{1}{129}$. The encoding and updating of probabilities continue until the end of the input stream. When decoding, the decoder also begins with the same starting probabilities and updates the distribution in the same manner as the encoder as each new symbol is decoded. Therefore, the encoder does not transmit probability distributions to the decoder.

The model just described is a *zero-order* model that assumed each symbol appears independent of other symbols. However, it is likely that the current symbol depends on the previous few symbols. This is true for English texts, where a letter *e* is likely to be preceded by a *h* due to the high frequency of the word *the*. Therefore, some statistical compression algorithms use *k*-th-order models, where the current symbol is encoded using the probability of that symbol occurring given the previous *k* symbols. The last *k* symbols are known as the *context*, and the larger the context, the better the predictive power of the next symbol, and the better the compression. However, *k*-th-order models are expensive to compute for large values of *k*. An implementation of a statistical compressor is *Prediction by Partial Matching (PPM)* [Cleary and Witten, 1984].
Statistical compressors generally perform better than dictionary compression methods in terms of the compressed size. Since a symbol is encoded based on its probability of occurrence, the number of bits used per symbol is close to the optimal number of bits suggested by the entropy. However, the probabilities of contexts cannot be maintained for large collections, hence repeats occurring across large distances will not be detected. Also, the lengths of repeats detected are restricted to the context size, unlike dictionary compressors that can detect arbitrarily long repeats. Statistical compressors also tend to be slow to compress and decompress, and can be memory-inefficient when the context is large. It is also difficult to support random access queries in data compressed with these algorithms, since the data must be decompressed until the random access position to access the probability distribution before decoding the portion of interest.

The use of statistical compression for DNA have been explored, and the best single-sequence compression results are achieved by one such compression tool. We evaluate the performance of some existing statistical DNA compression tools in Chapter 4. Next we discuss compression based on the Burrows-Wheeler transform.

### 3.2.4 Burrows-Wheeler transform

The Burrows-Wheeler transform or BWT [Burrows and Wheeler, 1994] is a reversible permutation of a string $T$ that groups together symbols based on their contexts. A tool such as `gzip` can compress the transformed string better than the original string because repeated symbols are localised to be within the sliding window.

Let $T$ be a string terminated with the special symbol $\$$, as for the suffix tree/array construction introduced in Section 3.1.3. The symbol $\$\$ is lexicographically less than all other symbols in the alphabet $A$ of $T$. The Burrows-Wheeler transform of a string $T$ is constructed as follows:

1. Form a matrix $M_T$, where each row of $M_T$ is a cyclic shift of the string $T$.
2. Sort the rows in $M_T$ in lexicographically increasing order.
3. The transformed text $L$ is the symbols in the last column of the sorted $M_T$.

An example of the BWT for the text $T = BANANA\$$ is illustrated in Figure 3.3. Note that all the occurrences of the symbol $N$ are now consecutive and all but one of the occurrences of the symbol $A$ are also consecutive. The cyclic shifts are sorted to group together the suffixes with the common prefixes, which also groups together the many occurrences of the same symbol provided they have the same contexts. The string $L$ is now more compressible than string $T$ [Burrows and Wheeler, 1994; Fenwick, 1996; Manzini, 2001] and $L$ can be compressed with any compressor.

Two data structures are required to reverse the BWT to obtain string $T$:
Figure 3.3: The BWT of the text $T = BANANA$$. The first column is the matrix containing all the cyclic shifts of the string $T$. The second column is the lexicographically sorted cyclic shifts of $T$. The last column is the BWT of the string $T$, which is the last column of the sorted cyclic shifts.

- An array $C$ of length $|A|$, where each $C[i]$ for $0 \leq i < |A|$, contains a cumulative sum of the symbols in $T$ that are lexicographically smaller than $A[i]$.

- A data structure that implements the query $rank(c, q)$, that returns the number of occurrences of the symbol $c$ in the prefix $L[0\ldots q]$.

Let $F$ be the first row of the sorted $MT$. Then the symbol at $L[i]$ immediately precedes the symbol at $F[i]$ in the original text. A method called Last-to-First column mapping or LF-mapping is used to determine the position $j$ in $F$ where symbol $L[i]$ occurs. This is calculated for position $i$ as:

$$LF(i) = C[L[i]] + rank(L[i], i)$$  \hspace{1cm} (3.3)

Intuitively, $C[L[i]]$ gives the position in $F$ where the occurrences of the symbol $L[i]$ begins, and $rank(L[i], i)$ gives the offset for the particular occurrence of the symbol $L[i]$. Such a mapping is possible because the $j$th occurrence of a symbol $a$ in $L$ corresponds to the $j$th occurrence of that symbol in $F$.

If the symbol at $T[k]$ is at $L[i]$, then symbol $T[k]$ is at $F[LF(i)]$. Since the symbol $L[i]$ precedes $F[i]$ in the original text, $L[LF(i)]$ must contain the symbol $T[k - 1]$. Using this property, and the data structures for $C$ and $rank(c, q)$, the original text can be reconstructed from $L$ as follows. First let $T[n-1] = L[0]$, since $L[0]$ is the last symbol of string $T$ because the first row of the sorted $MT$ is the cyclic permutation with the $\$ symbol at the front. Let $i = LF(0)$ and $k = n - 2$. Continue to assign $T[k] = L[LF(i)]$ and update $i = LF(i)$, until $k = 0$. At the end of this process, string $T$ is constructed in reverse.

The compression tool $bzip2$ is based on the BWT. The tool constructs a BWT for each block of the input string and uses it to compress the input. The block size can range between 100-900 Kbytes, depending on the chosen compression speed (100 Kbytes for fast compression). The transformed blocks are then encoded using a combination of encoding techniques. The tool is generally slow to compress but produces good compression results. We test the performance of $bzip2$ in Chapter 4 for compressing DNA sequences. In Section 3.4.1, we discuss compressed indexes based on the BWT. Next we discuss encoding techniques used by compression algorithms.
3.2.5 Encoding techniques

The final step of compression is to encode the symbols as efficiently as possible, according to the models determined in the second step of compression. This involves assigning codewords (bit patterns) for each distinct symbol, whose length depends on the probability of occurrence of that symbol. Generally, symbols with a higher probability receive shorter codewords, since they are likely to appear more frequently, while symbols with lower probabilities receive longer codewords.

Using the probability distribution, it is possible to generate near optimal compression; close to 0-order entropy if no context is used and close to $k$th-order entropy if a context of $k$ is used. Dictionary compression methods group together many base level symbols such that there is little or no repetition of symbols. Therefore, it is possible to assume that the symbols have equal probabilities and assign equal length codewords to each symbol. But better compression can be achieved by generating codewords based on the actual probability distribution. Statistical compressors use variable length codewords based on the probability distribution of symbols, and frequently use arithmetic coding to generate codewords. Dictionary compressors generally tend to use Huffman coding.

During compression, symbols that are not part of the compression model needs to be encoded so that the decompressor has enough information to decode the compressed data. Integers are commonly required to represent lengths and other identifiers. For example, integers that specify the lengths of substrings on the right-hand side of substitution rules are required to inform the decompressor about the number of symbols to be decoded for a given non-terminal. Generally, such information is not part of the model, hence alternative techniques are required to encode this information. For encoding a set of integers within a certain range $[0\ldots j]$, the simplest method would be to assigned equal-length codewords of $\lceil \log j \rceil$ bits each. However, if $j$ is large, then many bits are required to encode each integer, which is wasteful if smaller integers are more frequently observed. While it is possible to determine a probability distribution for these integers, it is time consuming and unnecessary. Techniques such as Golomb coding, Elias-delta coding and Elias-gamma coding are available for encoding integers, which we discuss later in this section.

**Huffman coding** The Huffman coding technique was introduced by Huffman [1952]. To Huffman code an input, the probability distribution of the symbols to be encoded is required to construct a Huffman tree [Witten et al., 1999]. First, the distinct symbols and their probabilities are added to the leaf nodes of the tree. Then, the two leaves with the smallest probabilities are chosen and a parent node is created containing the sum of the two probabilities. Then the next two smallest probabilities are chosen and a parent node is created containing the sum of those two probabilities. The process continues until there is just one node without a parent.
3.2. GENERAL-PURPOSE COMPRESSION

Figure 3.4: The Huffman tree generated from the text “this is an example of a Huffman tree”. Each leaf node consists of the symbol and its frequency. Each internal node contains the sum of the frequencies of its two child nodes. The table shows the symbols, their frequency counts and the codeword generated for each symbol from the Huffman tree.

The result is the Huffman tree, and for each internal node, following the left branch outputs a 0 bit and following the right branch outputs a 1 bit. The path through the Huffman tree from the root to the leaf that contains the symbol of interest, specifies the codeword for that symbol. Once the codewords are determined, the input stream of symbols can be encoded. The process of creating the Huffman tree is illustrated in Figure 3.4.

Huffman coding generates prefix-free codewords, where no symbol has a codeword that is a prefix of the codeword for another symbol. Therefore there is no ambiguity, and when decoding, the Huffman tree can be traversed until a leaf is reached to determine the symbol represented by the codeword. Huffman coding is fast provided the probability distribution is available. The probability distribution can be determined dynamically but this is expensive. A drawback of Huffman coding is that the Huffman tree needs to be transferred to the decoder, since the decoder cannot infer the codewords. Canonical Huffman coding on the other hand only transfers the symbols ordered by codeword length, which is sufficient for the decoder to infer the codewords. We discuss Canonical Huffman coding next.

Canonical Huffman coding Canonical Huffman coding [Hirschberg and Lelewer, 1990] consists of three steps. First, the length of each symbol is determined. Then, the number of symbols for each codeword length is counted. Finally, a starting symbol is selected for each codeword length, and a codeword is assigned for this symbol. Then for a given codeword length, the symbols with that codeword length receive consecutive codewords beginning with the codeword for the starting symbol.

---

2Derived from the image Huffman Tree 2 at en.wikipedia.org/wiki/File:Huffman_tree_2.svg created by the user Meteficha, Oct 2007.
of that length. The encoder only transmits the list of symbols ordered on codeword lengths to the decoder. The decoder infers the starting codeword for each distinct length and stores it in an array. When decoding a symbol, the decoder accesses the list of starting codewords to determine the codeword that is closest but lower than the current codeword. Then it calculates the offset from the selected starting codeword to the current codeword, and uses this offset to access the symbol from the sorted list. This decoding method is more efficient than using an explicit Huffman tree, since the number of random memory accesses are reduced and there is no cost associated with storing the tree.

In the first step, to determine the codeword lengths, a heap-based algorithm is used where the frequencies of each symbol are used to determine the codeword lengths without explicitly assigning codewords to each symbol [Witten et al., 1999]. For determining the starting codewords for each length in the final step, starting from the lowest length, a codeword is created by multiplying the last used codeword for the previous length by two. This ensures that the codewords for the current length are not a prefix of any other codewords for symbols with lower code lengths.

Canonical Huffman coding has the same compression performance as simple Huffman coding, since codewords are assigned based on their probability of occurrence. The method is suitable for encoding symbols generated by static and semi-static algorithms, where the modelling step is completed before the encoding step. However, in adaptive compression, modelling and encoding are conducted simultaneously during a single pass through the input, so codewords must be generated dynamically. Arithmetic coding is therefore more suitable for adaptive compression.

**Arithmetic coding** Arithmetic coding [Rissanen, 1976] encodes the input stream of symbols by representing it as a rational number in the interval \([0, 1)\), by considering the probability distribution of symbols as each symbol is read. Initially, equal probabilities are assigned for each symbol, and the interval \([0, 1)\) is divided into sub-intervals based on the probability distribution, and a distinct interval is assigned to each symbol. For example, for the alphabet and probabilities \(\{a = 0.25, b = 0.25, c = 0.25, d = 0.25\}\), \(a, b, c\) and \(d\) are assigned the intervals \([0.0, 0.25), [0.25, 0.50), [0.50, 0.75),\) and \([0.75, 1.0)\), respectively. Then if a new symbol \(a\) arrives, the probability distribution is updated to reflect the arrival of the new symbol, and the interval \([0.0, 0.25)\) is further divided into sub-intervals based on the updated probabilities. This process continues until all of the input symbols are read. The illustration in Figure 3.5 shows how the string “acd” is arithmetic coded.

The encoding is a binary representation of a rational number from the final interval reached. The number is chosen such that it could be represented precisely as a binary fraction. The decoder has the same knowledge about the probability distribution as the encoder. Therefore, at any given time, the rational number belongs to a single interval, and the symbol represented by that interval is output.
Figure 3.5: An example arithmetic coding of a string “acd” with a starting probability distribution of \( \{a = 0.6, b = 0.2, c = 0.1, d = 0.1\} \). The highlighted intervals indicate the interval selected as each symbol is read. In the final interval, the dot represents the rational number chosen to encode the input string, 0.538.³

Then the sub-intervals are created based on the new probability distribution. The rational number once again belongs to only one interval and the symbol for that interval is output. The process continues until all the symbols are decoded.

Arithmetic coding is able to encode the input stream close to its entropy. The size of the final interval is the product of the probabilities of the symbols from the input stream. The logarithm of the final interval size is equivalent to the sum of the logarithms of each individual probability. Therefore, a symbol \( s \) of probability \( Pr[s] \) consists of \(-\log Pr[s]\) bits of the output, which is equivalent to the information content of the symbol [Witten et al., 1999]. However, in practice a few more bits are required to represent the rational number as a binary fraction and for byte-aligning the codeword. Arithmetic coding is generally slower than Huffman coding but is more memory efficient. It is ideal for encoding the output of statistical compression methods. One drawback is that arithmetic coded collections cannot support random access, therefore Huffman coding is more suitable if random access is required.

**Unary coding**  
Unary coding is the simplest of integer encoding techniques. An integer \( x \geq 1 \) is encoded by \((x - 1)\) 1 bits, followed by a 0 bit. For example, the number 5 is encoded as 11110. The decoder simply counts the number of 1 bits until a 0 bit is reached to obtain the original integer. Unary coding is effective for encoding small positive integers but it is too costly to encode larger integers.

**Elias-gamma coding**  
Elias-gamma coding [Elias, 1975] represents an integer \( x \geq 1 \) by a unary code for the number of bits required to binary encode \( x \), followed by

³Derived from the image Arithmetic Encoding at en.wikipedia.org/wiki/File:Arithmetic_encoding.svg created by the user Dcoetzee, Apr 2007.
x encoded in binary, with the highest power of two in x subtracted from it. For example, the integer 9 requires four bits to encode it. The unary code for the number of bits is 1110. Then \(9 - 2^{|\log_2 9|}\) is encoded using \(|\log_2 x|\) bits in binary form, which is 001. Therefore, 7 bits are used to encode 9 as 1110001. To decode, the number of 1 bits is counted until a 0 bit is reached. One more than the number of 1 bits is the number of bits required to encode the number \(x\), which is \(c_u = |\log_2 x|\) bits. Then \(c_u - 1\) more bits are read to obtain the number \(c_b\), and \(2^{c_u} + c_b\) is the decoded value of \(x\). Elias-gamma coding requires \(1 + 2|\log_2 x|\) bits to encode an integer \(x \geq 1\). The technique is more bit efficient for smaller integers. Therefore, it is useful in situations where mostly small integers are encoded, and the largest integer is not known in advance.

**Elias-delta coding**  Elias-delta coding [Elias, 1975] is similar to Elias-gamma coding, except, an Elias-gamma code is used to represent the number of bits required to binary encode \(x\). So to encode the integer 9, \(|\log_2 9| = 4\) is encoded as 1100, followed by 001 to produce the bit encoding 1100001. This representation requires one extra bit compared to an Elias-gamma coding. For smaller integers, Elias-delta coding may require more bits, but for larger integers it requires less bits than Elias-gamma coding. Overall, Elias-delta coding uses \(1 + 2|\log_2 \log_2 2x| + |\log_2 x|\) bits. Decoding is very similar to decoding in Elias-gamma coding. Instead of unary decoding the number of bits required to binary encode \(x\), \(1 + 2|\log_2 \log_2 2x|\) bits need to be Elias-gamma decoded.

**Golomb coding**  Golomb coding [Golomb, 1966] encodes an integer \(x > 0\) by dividing \(x\) by a constant \(b\), and unary encoding the quotient and binary encoding the remainder. For example, supposing \(b = 4\), the integer 9 is encoded as the bit sequence 10 for the quotient \(2 = \lfloor9/4\rfloor\) and a 1 bit for the remainder \(1 = 9\%4\). The final encoding is 101, which uses just 3 bits. The quotient uses \(q\) bits, which may be a large number of bits depending on how large \(x\) is with respect to \(b\). The remainder uses \(|\log_2 b|\) bits. To decode, the unary component is decoded to obtain \(q\). Then \(|\log_2 b|\) bits are decoded to obtain \(r\). Then \(x\) is calculated as \(q \times b + r\).

Evidently, the number of bits required to encode an integer is dependent on the value of divisor \(b\). For efficiency in implementation, powers of two are chosen for the value of \(b\) so that bit shifting can be used to find the quotient and the remainder rather than the more expensive divide and modulo operators. If a small \(b\) value is chosen, then for small integers, a small number of bits will be used whereas large integers will require a larger number of bits, and vice versa for a large value of \(b\). Therefore, a small value of \(b\) could be chosen if the set of integers to be encoded is expected to mostly have small values, and a larger \(b\) if large integers are expected. As a rule of thumb, if a probability of occurrence of a given integer satisfies \(p = \frac{f}{N} \ll 1\), where \(f\) is the expected number of occurrences of the integer \(x\), and \(N\) is the total
number of integers to encode, then $b \approx 0.69 \times \frac{N}{f}$.

In this section, we discussed several types of general-purpose compression algorithms. These algorithms and their implementations have been designed for general use without making assumptions about the types of data that will be compressed. However, by tailoring a compression algorithm to compress a particular type of data, it may be possible to achieve better results. Therefore, specialised compression algorithms for DNA may be necessary to efficiently store the data produced by DNA sequencing projects. In the next section, we discuss the motivation for DNA-specific compression, and discuss some existing DNA compression algorithms. In Chapter 4, we evaluate the performance of general-purpose and DNA compression algorithms for compressing several DNA sequences and collections.

### 3.3 DNA compression

Using fixed length codewords, a DNA sequence containing nucleotides of the standard four symbol alphabet $\sigma = \{a, c, g, t\}$ can be represented using just 2 bpb. Assuming that each nucleotide occurs with equal or close to equal probabilities, the 0-order entropy is also 2 bpb, making such an encoding optimal. However, sequences tend to have biases towards certain nucleotides depending on the species or the function of the sequence. For example, the $P. falciparum$ genome tends to contain a higher proportion of $\{a, t\}$ nucleotides. Therefore, variable length codewords can be assigned to nucleotides based on the probability distribution to achieve a near-optimal encoding. Nucleotides from the extended alphabet $\Sigma$ can also be present in DNA sequences, especially the nucleotide $n$, but they occur rarely, so sequences can be encoded with an average of 2 bpb. We can expect general-purpose compression tools to encode DNA sequences to 2 bpb or better if repeats are taken into account. However, some of these tools tend to increase the size of the compressed sequences above that of the fixed-length encoding [Grumbach and Tahi, 1993].

Since repeats in DNA sequences are far apart, the sequences may appear to be random strings within the small window used in dictionary based compression models, or in context-based statistical compressors, which may explain the lack of compression. Also, as discussed in Section 2.1.4, DNA sequences contain reverse complement repeats and approximate repeats, which are DNA-specific properties that will not be detected by general-purpose compressors.

Therefore, compression algorithms that detect repeats specific to DNA may be required to achieve better compression. Grumbach and Tahi [1993] made this observation and introduced the first DNA-specific compression algorithm, Biocompress. Since then, many other DNA compression algorithms were invented, and the concepts used in general-purpose compression algorithms formed the basis for these new algorithms. Most use LZ77-type dictionary compression techniques, while others use statistical compression techniques. Some DNA compression algorithms detect exact
repeats, while others detect both exact and approximate repeats. We divide our discussion based on these two repeat detection techniques.

### 3.3.1 Exact matching

The algorithms discussed in this section compress DNA sequences by detecting exactly repeated substrings and encoding them efficiently. Most of these algorithms apply the same techniques as well-known general-purpose compression algorithms such as LZ77. Modifications are made to allow the algorithms to perform better on DNA sequences. **Biocompress** [Grumbach and Tahi, 1993] was the first DNA-specific compression algorithm. Below we briefly discuss **Biocompress**, its successor **Biocompress-2**, and four other exact repeat detecting algorithms.

**Biocompress and Biocompress-2 algorithms** Grumbach and Tahi [1993] were the first to explore DNA-specific compression, as they realised the potential onset of DNA sequence collections with many repetitive sequences. The **Biocompress** algorithm is based on LZ77 (Section 3.2.2), where repeated substrings are represented as references to an earlier occurrence of the substring in the input. Recall from Section 3.2.2 that the sliding window size was a restriction of LZ77 implementations that prevents repeats that are far apart in DNA sequences from being detected. To avoid this, **Biocompress** stores all of the so far seen input in memory using a 4-ary tree of height $h$. The leaf nodes at depth $h$ contains positions at which the substring represented by the path from the root node to the leaf node occurs in the input.

When a longest matching substring is found, the algorithm encodes the matching substring as a factor if encoding a factor uses less bits than encoding the substring using 2 bpb. Otherwise the current base is encoded as a single literal and the algorithm attempts to find the longest matching substring starting at the next position in the input. **Biocompress** also supports reverse complement repeat detection.

Grumbach and Tahi [1993] used their implementation of the **Biocompress** algorithm to compress a set of DNA sequences. The results showed that **Biocompress** compressed the test sequences to a smaller size than expected from a simple 2 bpb encoding, compared to the general-purpose algorithms **compact** and **compress**, which used more than 2 bpb on average. This result shows that DNA compression algorithms have an advantage in compressing DNA sequences compared to general-purpose compression algorithms, and led the way to a new area of research in improving algorithms that compress DNA sequences.

**Biocompress-2** [Grumbach and Tahi, 1994] improves upon the compression results of **Biocompress** by allowing for runs of single literals to be encoded using an order-2 arithmetic encoder. The use of an encoder that produces codewords based

---

4. **compact** is an obsolete Unix compression tool that uses adaptive Huffman coding to encode the symbols but as a result was very slow [Witten et al., 1999].
on the probability distribution of symbols produced promising results. For some DNA sequences in the test dataset, the compressed size more than halved.

Grumbach and Tahi [1993] also realised that DNA sequence collections can contain highly similar sequences, such as genomes from the same species or genes serving the same function from various species. Therefore, it is possible to conduct “vertical” compression, where the sequences in a collection are compressed against a reference string that represents the remaining sequences in the collection. Biocompress-2 was used to conduct vertical compression, where the ribosomal RNA sequences of a few bacterial species was compressed in pairs using one of the sequences in the pair as a reference to compress the other. The results showed that, the closer the species are phylogenetically, the better the compression results. We explore vertical compression in Chapter 6, where we introduce an LZ77-type compression algorithm that compresses collections of highly similar sequences against a reference.

Cfact algorithm Cfact [Rivals et al., 1996] is similar to Biocompress, and uses a suffix tree of the input sequence to detect repeats. The algorithm operates in two phases. The parsing phase constructs a suffix tree of the input sequence and uses it to select the most significant repeats. A significant repeat is one that can be encoded to obtain a local compression gain. In the encoding phase, the input sequence is read from left to right, and the non-repetitive substrings (the substrings that were not identified as significant repeats) and the first occurrence of repetitive substrings are encoded using 2 bpb. The second occurrence of a repeated substring is encoded as a reference to the first occurrence. Any subsequent occurrences are encoded as an index that identifies the repeat. The algorithm operates in $O(n^2)$ time and space.

The algorithm ensures that if there are no repeated substrings in the input sequence, the compressed size will not exceed the size of the basic 2 bpb encoding. We could not find compression results for the algorithm, therefore it is difficult to compare Cfact with Biocompress or the other general-purpose compression tools. We expect Cfact to produce similar compression results to Biocompress, since both algorithms find the longest repeated substrings and encodes it in an LZ77 manner. However, we also expect that Cfact is slower to compress than Biocompress, since two passes over the input is required, and constructing a suffix tree is more time consuming than constructing a 4-ary tree.

The main drawback of the algorithm is that for a large collection of DNA sequences, it is expensive to store a suffix tree in memory, as discussed in Section 3.1.3. Therefore, an alternative data structure such as a compressed suffix tree, a suffix array or a compressed suffix array can be used. In Chapter 8, we explore a variant of this algorithm to detect repeats in a collection to construct a reference string.

dna-x algorithm Apart from Biocompress and Cfact, most algorithms until dna-x detected approximate repeats. However, these were slow to compress and
their memory requirements made it infeasible to compress large sequences. Therefore, Manzini and Rastero [2004] re-introduced exact repeat detection with the \texttt{dna-x} algorithm, paying special attention to the memory use of the algorithm. For a sequence of length $n$, \texttt{dna-x} uses $(7n)/5$ bytes of working space. This allowed \texttt{dna-x} to compress the largest sequence compressed until that time, which is a 220 MB \textit{H. sapiens} chromosome. Manzini and Rastero [2004] claimed that \texttt{DNACompress}, the best performing algorithm at the time, uses $6n$ bytes of working space.

Manzini and Rastero [2004] discussed three important repeat properties to detect; repeats that are far apart in the input, reverse complement repeats and approximate repeats. The \texttt{dna-x} algorithm explicitly detects the first two types of repeats but the third kind of repeats is detected and encoded implicitly. It is a single-pass algorithm, and operates in an \textit{LZ77} manner by finding substrings that occurred earlier in the input to the current position being encoded. The dynamic dictionary is implemented differently. For a length $B$, every $B$th non-overlapping substring is recorded in a hash table $H$. If $i$ symbols are encoded so far from input $M$, then the substring $M[i \ldots i + B]$ is checked in $H$. If that substring is not found, then $M[i + 1 \ldots i + B + 1]$ is checked. The process continues, until a substring $M[i + l \ldots i + B + l]$ is found in $H$ at a position $p$ in $M[0 \ldots l]$. Then the two matches are extended backwards from positions $p$ and $i + l$ and forwards from positions $p + B + l$ and $i + B + l$, to find the longest possible match. After the longest match is found, two factors are encoded; the non-matching component and the matching component. Using this process, all repeats with a length of at least $2B - 1$ and some repeats with lengths in the range $B$ and $2B - 2$, can be encoded. No repeats with a length $< B$ will be detected.

The algorithm is memory efficient since only $n/B$ items will be stored in the hash table. By increasing the $B$ value less memory will be used at the cost of less of the repeats being detected. The authors explored various encoding strategies for encoding the non-matching component and the matching component. The compression results of \texttt{dna-x} using the best encoding strategies is better than the results of general-purpose compression tools such as \texttt{gzip} and \texttt{bzip}. While the authors attempted to compare \texttt{dna-x} to existing DNA compression tools, most of these were unavailable, and those that were available could not compress the larger sequences. For compressing a set of short sequences, the approximate repeat detection algorithms like \texttt{GenCompress}, \texttt{CTW+LZ} and \texttt{DNACompress} performed better than \texttt{dna-x}, but \texttt{dna-x} results were only slightly worse.

The \texttt{dna-x} algorithm shows that DNA-specific compression algorithms that detect only exact repeats provide the best trade-off between compression and resource usage. We conduct our own comparison of \texttt{dna-x} to existing general-purpose and DNA compression algorithms in Section 4. Then in Chapter 8, we use \texttt{dna-x} to test its ability to generate a dictionary of repeats.
DNASequitur algorithm DNASequitur [Cherniavsky and Ladner, 2004] is a DNA-specific variant of the Sequitur algorithm described in Section 3.2.2, that outputs a context free grammar of the input. The novelty of DNASequitur is its ability to recognise reverse complements when creating rules and during substitutions. The results show that DNASequitur outperforms Sequitur by producing a slightly lower number of rules for DNA sequences. The total number of symbols on the right-hand side of a rule is lower with DNASequitur, the number of symbols in the longest rule is greater and the most frequently used rule has a higher frequency than in Sequitur. The authors explored various encoding techniques to represent the grammar. After using their best performing encoder, and improving the grammar with techniques such as removing rules that expand to form the same substring, the algorithm did not out-perform the best DNA compression algorithm at the time, DNACompress.

Given that the authors had improved Sequitur in several ways to support DNA compression, the results produced by DNASequitur led the authors to conclude that exact repeat detection and grammar compression are not suitable for compressing DNA sequences. Exact repeat detection may be unsuitable for compressing DNA due to the presence of mutations. Grammar compression may also be unsuitable for compressing single DNA sequences, since single sequences tend to be non-repetitive, as discussed in Section 2.1.4. However, grammar-based compression is ideal for detecting repeats that occur across multiple sequences in a collection, as will be shown in Chapter 5. Another advantage of grammar-based compression is the ability to support random access and search in the compressed collection. Therefore, grammar-based DNA compression should not be ruled out.

Although exact repeat detection is efficient, it is not appropriate for single-sequence compression, since mutations are not taken into account. Therefore, most DNA compression algorithms are based on approximate repeat detection, which improves compression at the cost of higher memory use and longer compression and decompression times. Next we discuss DNA compression algorithms that use approximate repeat detection.

3.3.2 Approximate matching

As discussed in Section 2.1.4, some repeats in DNA sequences, especially the repeats within sequences, are likely to contain mutations. Even collections of sequences from evolutionarily close species may have mutations that have accumulated since the divergence of the species from their common ancestor. Therefore, approximate repeat detection is useful for compressing single sequences and certain sequence collections. Most DNA compression algorithms implement approximate repeat detection as a result. First we discuss some algorithms which are based on LZ77, followed by some statistical compression algorithms.
GenCompress algorithm  GenCompress [Chen et al., 2000] follows the LZ77 process, where a repeated substring is encoded as a reference to an earlier occurrence. The difference is that the repeated substring could contain mutations, and these mutations are encoded with the factor. The mutations could be single point mutations, insertions or deletions. The number of mutations permitted in a match are restricted to narrow the search space and to ensure that compression is effective. A factor also has to satisfy a gain function, which measures whether the number of bits used to encode the factor is less than encoding the substring itself.

In the experimental results of Chen et al. [2000], GenCompress performed better than Biocompress-2 overall. GenCompress was also compared to Cfact, where it consistently achieved better results. Even after using two passes and detecting repeats globally, Cfact was unable to perform as well as GenCompress, which only uses one pass through the input to compress. This showed that approximate repeat detection is very effective for single-sequence compression.

The results however do not discuss the compression speed or memory use of the algorithm. The largest sequence compressed contained only just over 200 Kbases, which is a small sequence compared to large genomes such as the human genome. The results presented by the authors of DNACompress and our own results in Chapter 4 shows that GenCompress is indeed slow to compress sequences as small as 1 Mbase in length. While GenCompress produces good compression results, it is not suitable for compressing large sequences or collections.

DNACompress algorithm  DNACompress [Chen et al., 2002], in a similar manner to GenCompress finds approximate repeats and encodes them in an LZ77 manner, along with mutation information. Approximate repetitions are detected using the PatternHunter homology search tool. The algorithm has a three step process. First, the input sequence is given to PatternHunter to find all the approximate repeats, which are returned sorted in descending order of similarity score. DNACompress then selects repeats from highest to lowest score that does not overlap with any repeats that are chosen already. This process continues until a minimum score threshold is reached. In the final step, repeats and non-repeats are encoded. The repeats selected in the second step are encoded as LZ77 factors with mutation information. Repeats are only encoded as factors if the encoding cost of the factor is less than the cost of encoding the original substring.

DNACompress was compared to compress, GenCompress and CTW+LZ algorithms. The results showed that DNACompress outperformed compress but only produced slightly better results than the other two algorithms. The biggest improvement made by the algorithm was in terms of compression speed. DNACompress was able to compress even the largest sequence (Escherichia coli with over 4.5 Mbases) in under a minute, where GenCompress required nearly half an hour to compress the sequence, while CTW+LZ was not tested, since it would require many hours. For many
3.3. DNA COMPRESSION

years, DNACompress was regarded as the best compression algorithm.

**DNA Pack algorithm** GenCompress detects repeats greedily, where the next longest matching prefix is encoded without looking ahead to check if there is a longer match further ahead that results in a better compression result. DNACompress resolves this issue by first detecting all repeats, and then selecting the “best” repeats to be encoded by favouring non-overlapping repeats that has a high similarity score. However, it may be better to encode two repeats that overlap with the chosen repeat, that have a total length greater than that of the chosen repeat. Behzadi and Le Fessant [2005] claimed that with a dynamic programming calculation, a better set of repeats than those found by GenCompress and DNACompress could be found.

The dynamic programming calculation considers the cost of encoding a substring as a repeat (including edit information for approximate matches), as a reverse complement repeat, or as a non-repeat. Therefore, the complexity of the calculation is $O(n^3)$ for a sequence of length $n$. This cannot scale even for reasonably large sequences, so several optimisations were introduced to reduce the cost. The authors also focused on making the encoding more efficient. The algorithm was compared to Biocompress-2, GenCompress, CTW+LZ and DNACompress in terms of the compressed size. DNAPack consistently performed better than the former three algorithms, while it also compressed better than DNACompress for most sequences. The authors did not discuss the compression speed or memory use of their implementation. We expect that the algorithm will be slow due to the expensive calculation, and will not be suitable for compressing large sequences. We could not find an implementation to test our hypothesis. Next we discuss three statistical DNA compressors.

**CDNA algorithm** CDNA [Loewenstern and Yianilos, 1997] was the first algorithm to combine statistical compression with approximate repeat detection for DNA compression. The probability distribution of each nucleotide is predicted by approximate partial matching of the current context to previously seen substrings, using a Hamming distance to measure the similarity. The model is predicted using a test dataset prior to encoding the complete sequence. The performance of CDNA was evaluated for compressing DNA sequences from various species, and from coding and non-coding regions. The performance in terms of the compressed size was compared to Biocompress-2 and compress. CDNA performed better than compress as expected, and also outperformed Biocompress-2 for most sequences. Once again, there was no discussion of the compression speed or the memory use of the implementation. It is difficult to assume that with approximate matching, CDNA will be able to efficiently compress large sequences or collections. We attempt to compress some large sequences using CDNA in Chapter 4. Another issue is that CDNA has many model parameters which do not have any biological interpretations [Cao et al., 2007], making it difficult to determine the parameters that should be used to achieve a good result.
**CTW+LZ algorithm**  
CTW+LZ [Matsumoto et al., 2000] combines both dictionary and statistical compression techniques. Long repeats are encoded using an LZ77-type encoding. Short repeats and non-repeats are encoded using a *Context Tree Weighting* (CTW), which is similar to PPM, but uses a weighting of multiple models to determine the next symbol probabilities. The algorithm detects approximate repeats using dynamic programming. If a repeat including the edit operations require more bits to encode using LZ77 than using a CTW encoding, then it is encoded using CTW, and otherwise using an LZ77 encoding. By combining the strengths of both dictionary and statistical compression techniques, one can expect CTW+LZ to perform well. The algorithm tends to perform better than Biocompress−2 and GenCompress. The major disadvantage of the algorithm is that it is extremely slow to compress as shown by Chen et al. [2002], where a sequence of just 229 Kbases required many hours to compress. Therefore CTW+LZ will not scale for compressing large sequences.

**NML and GeNML algorithms**  
NML [Tabus et al., 2003] and its successor GeNML [Korodi and Tabus, 2005, 2007], introduced an alternative statistical approach to compressing DNA. NML [Tabus et al., 2003] separates the input into fixed-size blocks. To encode a block, the algorithm finds a “regressor”, which is a substring that occurred earlier in the input that has the minimum Hamming distance from the current block, and encodes the block as a reference to the earlier occurrence and a bit mask to represent the differences between the current block and the regressor. The bit mask is encoded using an order-0 *Normalized Maximum Likelihood* (NML) model between the current block and the regressor. GeNML [Korodi and Tabus, 2005] allows the input to be separated into variable-size blocks, among other improvements to the encoding. GeNML was the first algorithm to compress a large sequence such as the 3 Gbase *H. sapiens* genome, achieving a compressed size of 1.535 bpb. Further improvements to GeNML include, using an order-1 NML model to encode the bit-mask, where the current bit is encoded based on the previous bit, and selecting an optimal regressor from a set of candidate regressors [Korodi and Tabus, 2007]. The improvements were achieved within the same time complexity, and the compressed size of the human genome was reduced to 1.510 bpb. However, both the above results were achieved with a compression time of 3.5 hours each, using a cluster of 12 workstations with 3.2GHz Pentium 4 processors. The NML and GeNML algorithms clearly produce good compression results. However, it is unclear that either of these algorithms can be scaled to compress large sequences or collections.

**XMCompress algorithm**  
XMCompress [Cao et al., 2007] is based on PPM. The difference is that multiple ‘experts’ make recommendations on the symbol probabilities rather than just using the context to predict the probability of a symbol. An example of an expert is an order-\(k\) Markov model that gives a probability of a symbol based on the \(k\) preceding symbols. Another is a copy expert that gives a probability
3.3. DNA COMPRESSION

based on whether the next symbol is part of a copied region. A related expert is the reverse copy expert, which is a copy expert for reverse complements. Although the probabilities from the experts are combined to make a single probability distribution, different weightings are given to different experts for different areas of the sequence, depending on the predictive power of the experts. In this manner, unreliable experts will contribute less to the probability than reliable experts.

When XMCompress is compared to existing tools, it yielded the best compression results for a set of short sequences. XMCompress also compressed a human genome and produced the best compression results known to date. The algorithm is excellent for single-sequence compression. The authors have also implemented a feature to compress a sequence with respect to a reference string. This gives even better compression results and is able to show areas of the two sequences that are shared using conditional information content results produced by the algorithm. Therefore XMCompress could even be used as a homology detection tool.

A key problem with XMCompress is that the statistical nature of the algorithm and the ability to detect approximate repeats mean that the compression is slow even for small sequences, making the algorithm impractical for compressing large sequences. In Chapter 4, we use XMCompress to compress our test datasets, which shows the excellent compression results and the poor compression speed.

Next we describe another type of compression algorithms for DNA, which only encodes the differences between sequences from the same species. The mutation rates in such sequences are small, so it is sufficient to just encode the mutations with respect to a reference string.

3.3.3 Compressing variations

Currently, much of the data released by high-throughput sequencing projects, such as the 1000 Genomes project, are in the form of variation data. The variation data contains SNP, insertion and deletion information for each genome, with respect to the reference string for that species. Algorithms that are specifically designed to compress variation data exist. Below we discuss two such algorithms.

The DNAzip algorithm by Christley et al. [2009] is designed for storing the variation data from the 1000 Genomes Project. The algorithm requires the H. sapiens reference genome and a database of known SNPs to be available to compress the variation data, which is 4.47 Gbytes in the test dataset provided with the implementation of the algorithm. The test dataset contains the James Watson genome variations which is 1.97 Gbytes in size initially, but after extracting the positions of occurrence of the mutations and the changes with respect to the reference genome, the data is only 84.5 Mbytes in size, significantly smaller than the original dataset. The extracted variation information is encoded as follows. The SNPs that are already present in dbSNP are not encoded, and a bit vector is used to determine which of
the SNPs occur in the data. The nucleotide of a new SNP is encoded using 2 bpb, and insertion nucleotides are encoded by Huffman coding \( k \)-mers. Deletions are represented as a position and a length. All positions are represented as a difference from the previous SNP, insertion or deletion position, since encoding the difference requires less bits than encoding the position itself. Using this method, the variation data for James Watson can be compressed to 4.2 Mbytes.

**DNAzip** is effective for compressing variation data for intact human genomes. It is unclear whether Christley *et al.* [2009] simply wanted to compress variation data, or the variation data is to be used for regenerating the original assembled genome. It should be noted that compressing the variations in a genome is not equivalent to compressing an assembled genome. The variation data available for projects such as the 1000 Genomes project are not generated by full assembly of the reads, and are determined by mapping the reads to the reference genome. As a result, other types of mutations that occur in genomes, such as large scale indels, rearrangements and translocations are not captured. To determine a complete set of variations, the reads need to be assembled and aligned to the reference genome. Assembly and alignment are both computationally expensive operations. If the aim is to compress a variation data file, then this is a lossy algorithm, where some information is discarded.

The current implementation can only compress human genome variation data, but the algorithm could be extended to compress variation data for other species with a reference genome and a SNP database. However, the method is limited to compressing sequences where the dissimilarities are predictable and limited. It would not be effective, for example, for deranged genomes such as those found in cancer, which can contain hundreds of megabases of arbitrarily repeated or rearranged material. Its effectiveness is due to the use of a highly specific model of the data that is anticipated, and, in contrast to a typical pattern-based compressor, it is limited to data that fits a predefined template. Apart from the above short-comings, the overhead required to compress or decompress is large. If a dataset compressed by **DNAzip** is to be shared among research groups, the versions of the reference genome and **dbSNP** database needs to be consistent. Otherwise decompression will be erroneous. If various research groups use different versions of these datasets, then the overhead is even more significant to maintain the different versions.

Moreover, **DNAzip** cannot be applied for compressing arbitrary collections of sequences from multiple species, which may not necessarily be complete genomes, or may even be unassembled contigs. Brandon *et al.* [2009] proposed a similar but more general technique. The aim is to allow any collection to be compressed by permitting a single or multiple references to be specified. The compression techniques used are similar to that of Christley *et al.* [2009]. Various encoding techniques are explored for encoding the difference in positions and variants.

In collections where the sequences originate from the same species or are variant sequences that serve the same biological function, the sequences will have a large
 proportion of common DNA. Therefore, large space savings can be made by just storing the differences with respect to a reference genome. The drawback of the work of Brandon et al. [2009] is that it does not consider the cost of determining the variation of each sequence in the dataset with respect to some reference. If this data is not available, or if a different reference is to be chosen, then each sequence in the dataset needs to be aligned to the reference to find the variations. Aligning sequences can be expensive, especially for large sequences or genomes. Therefore, this will be a large overhead of the compression process.

Another criticism of this type of compression is that a different encoding technique is required for each type of mutation, making the method ad hoc. Instead, it is better to have a more general algorithm that makes no assumptions about the types of mutations in the collection. We introduce our RLZ algorithm in Chapter 6, which compresses each sequence in a collection with respect to a reference, but no assumptions are made about the types of mutations.

In this section, we presented DNA-specific compression algorithms in the literature in terms of algorithms that detect exact repeats and those that detect both exact and approximate repeats. We evaluate the performance of some of the above-mentioned DNA compression algorithms in the next chapter, in terms of their performance for single-sequence compression and collection compression. Although variation data compression is also interesting, many improvements are required before these algorithms can be used for compressing more general DNA datasets. In this thesis we only focus on compression algorithms for assembled data. In the next section, we present some novel data structures that represent strings in compressed from, while supporting random access and search functionality on the compressed strings. These data structures have the potential to be useful for efficiently storing and accessing large DNA sequence collections.

3.4 SELF-INDEXES

Compression algorithms provide the means to store data compactly, and the more repetitions there are, the more effective the compression. Typically, a compressed datasets is in a form that is not usable, and needs to be decompressed before substrings can be accessed or searched. The decompression overhead can be high, especially for large datasets. Therefore, it would be ideal to compress the data in such a way that search and access queries can be supported efficiently on the compressed version. The \textsc{ray} [Cannane and Williams, 2001] algorithm supports substring retrieval from the compressed data.

Similar to \textsc{ray}, dictionary compression algorithms can be extended to support these queries on the compressed data. The first such attempt for the LZ77 algorithm was made by Kärkkäinen and Ukkonen [1996], which resulted in a $k$-mer index using a LZ77-compressed string. However, the original string was still required for certain
queries. The most promising advancement was made by Ferragina and Manzini [2000], who noticed the relationship between the Burrows-Wheeler transform (BWT) of a string and its suffix array. They introduced the notion of combining compression and indexing, and paved the way to a new area of research to implement search functionality on compressed datasets. An index built on a compressed dataset was named a self-index. Below we discuss some of the main self-indexes. First we discuss a few indexes based on the Burrows-Wheeler transform. Then we discuss an index based on a compressed suffix array, followed by LZ77 indexes.

3.4.1 Burrows-Wheeler index

Earlier, in Section 3.2.4, we introduced the Burrows-Wheeler transform, which transforms the string by grouping together the symbols of the string based on their contexts. This makes a string more compressible, since occurrences of the same symbol tend to be grouped together. One of the most interesting aspects of the BWT construction is that the suffixes of the string $T$ are sorted to obtain the BWT. The positions in $T$, for each symbol in $F$, constitutes the suffix array of $T$. Therefore, LF-mapping can be used to answer the count $(P)$ and locate $(P)$ queries. Ferragina and Manzini [2000] first noticed this interchangeability between the BWT of a string and its suffix array. They used this property to construct the first Burrows-Wheeler index known as the FM-Index, which paved the way to compressed indexes.

FM-Index The FM-Index [Ferragina and Manzini, 2000, 2005] of a string $T$ simply consists of the BWT $L$ of the string $T$, the cumulative symbol counts $C$ and the data structure that supports rank $(c, q)$ queries. To answer a count $(P)$ query, the backward search algorithm is used. First the range $l \leq i \leq r$ for $L$ is found, where $F[LF(i)]$ corresponds to the last symbol in the substring $P[m-1]$. Then the range $l$ and $r$ are narrowed as each symbol in $P$ from $k = m - 2$ until $k = 0$ is added to the search, where $m = |P|$. Then $count(P) = r - l + 1$. The algorithm preserves the invariant that at each $i$th iteration, $l \leq j \leq r$ is the range where $P[i \ldots m]$ is the prefix of each row $M_T[j]$. The time complexity of the count $(P)$ query is dependent on the time complexity of rank $(c, q)$. Ferragina and Manzini [2000, 2005] describe an algorithm that computes rank $(c, q)$ in $O(1)$ time so that count $(P)$ has $O(m)$ complexity. The size of the compressed $L$ and the associated data structures are bounded by $5nH_k(T) + O(n \frac{\log \log n}{\log n})$ bits for any $k \geq 0$.

Supporting locate $(P)$ is non-trivial, since $L$ and the other data structures have no information about the starting positions of each row in sorted $M_T$ (the suffix array). Therefore, sampling some of the positions is necessary, and these samples can be used to infer the positions. Firstly, the position in $T$ where symbol $F[r_j]$ occurs is stored in an array $S$, where $r_j$ is a sampled row number in sorted $M_T$. Let $Pos(i)$ be the position in $T$ where symbol $F[i]$ occurs. Row $r_j$ is chosen such that $Pos(r_j) = 1 + j\eta$ where $\eta = \lceil \log^{1+\epsilon} n \rceil$ (chosen depending on the desired memory
consumption, but higher the value of $\epsilon$, the smaller the space). Using the sampled positions in $S$, given a row index $i$, first check if there is a sampled position for row $i$. If it is sampled, then the position can be retrieved directly from $S$. Otherwise, use the backward search algorithm to determine the row $i'$ such that $\text{Pos}(i') = \text{Pos}(i) - 1$. This is continued until a row $i'$ is found where the position of that row is sampled in $S$. Then the position is obtained by $\text{Pos}(i) = \text{Pos}(i') + t$, where $t$ is the number of iterations taken to find a sampled row. Therefore, to answer a $\text{locate}(P)$ query, first obtain the range $l \leq i \leq r$ from a $\text{count}(P)$ query. Then for each row $i$, obtain the position at which the row starts in text $T$. The $\text{locate}(P)$ query can be answered in $O(p + \text{occ} \log^{1+\epsilon} n)$ time, where $0 < \epsilon < 1$ is the arbitrary constant chosen depending on the space usage of the samples. The space usage of the index is bounded by $5nH_k(T) + \frac{n}{\log n}$ bits for any $k \geq 0$. Ferragina and Manzini [2000] also describe an alternative, but more complicated representation that can answer a $\text{locate}(P)$ query in $O(p + \text{occ} \log^k n)$ time using $nH_k(T) + \frac{n \log \log \log n}{\log n}$ bits.

A $\text{display}(s,e)$ query is supported in a similar manner to that of $\text{locate}(P)$. Firstly get the sampled position that occurs either at or after position $e$. Then use LF-mapping until position $s$, while using the data structure that answers the $\text{rank}(c,q)$ queries to determine the symbol that occurs at each position between $s$ and $e$. Using the $\text{rank}(c,q)$ data structure proposed by Ferragina and Manzini [2000, 2005], a $\text{display}(s,e)$ query can be answered in $O(e - s + \log \epsilon n)$ time.

Most improvements made to the \textbf{FM-Index} are in the form of improving the space consumption and time complexity associated with the $C$ data structure and $\text{rank}(c,q)$ data structure [Navarro and Mäkinen, 2007]. On the other hand, the \textbf{RLFM} [Mäkinen and Navarro, 2005] and \textbf{RLWT} [Mäkinen et al., 2009; Mäkinen et al., 2010] self-indexes store the BWT $L$ of the text $T$ more efficiently, which allows the $\text{rank}(c,q)$ query to be answered directly using $L$.

The \textbf{FM-Index} and its variants \textbf{RLFM} and \textbf{RLWT} produce reasonable compression results and also have good query performance. However, one of the variants of the compressed suffix array self-index, which we describe next, has a better compression and query performance.

### 3.4.2 Compressed suffix arrays

A \textit{Compressed suffix array} or \textbf{CSA} [Grossi and Vitter, 2000] uses the redundancy in a suffix array to represent it in less space than $n \log n$ bits. The redundancy comes from \textit{self repetitions}, which are defined as a maximal interval $SA[i \ldots i + l]$ with a target interval $SA[j \ldots j + l]$, where $SA[j + r] = SA[i + r] + 1$ for $0 \leq r < l$. Intuitively, for a set of suffixes with a common prefix, the set of following suffixes for each item in this set will have the same ordering. For our example suffix array in Figure 3.2, the suffixes in $SA[5 \ldots 6]$ follow the suffixes in $SA[2 \ldots 3]$. If we were to define an array containing the positions in the suffix array where the following
suffixes are stored, then for intervals such as the ones described above, each position will be one more than the previous position. Let us define this array as $\Psi$, where $\Psi = \text{SA}^{-1}[$\text{SA}[$i] + 1], where $\text{SA}^{-1}[$i is the position where the $i$th suffix occurs in $\text{SA}$ [Mäkinen et al., 2009]. The $\Psi$ array for our example is $\Psi = 4, 0, 5, 6, 3, 1, 2$.

Note that $\Psi(i + 1) = \Psi(i) + 1$ when $\Psi(i)$ and $\Psi(i + 1)$ are in the same run, as at $\Psi[2]$ and $\Psi[3]$, and at $\Psi[5]$ and $\Psi[6]$. The advantage of storing $\Psi$ instead of $\text{SA}$ is that, unlike the random dispersion of integers in the range $0 \ldots n - 1$ in $\text{SA}$, the integers in $\Psi$ have predictable behaviour that can be used for compression. Due to the self-repetitions, there are intervals in $\Psi$ where $\Psi(i + 1) = \Psi(i) + 1$. Therefore, we can differentially encode $\Psi$ as $\Psi(i + 1) - \Psi(i)$.

Using $\Psi$ and the data structure $C$, where $C[c]$ gives the number of symbols in string $T$ that are alphabetically less than the symbol $c$, $\text{count}(P)$ queries can be answered in the same manner as in a standard suffix array using two binary searches. A bit vector $F$ can be used to store a 1 bit at each position in $\text{SA}$ where the first symbol of the suffixes change. At each step of the binary search, the substring $P$ needs to be compared with the prefix of the suffixes. To retrieve the prefix for $\text{SA}[i]$, $\text{rank}_1(F, i)$ gives the symbol of $T[\text{SA}[i]]$. Then $\Psi[i]$ gives the next suffix array position $j$ and $\text{rank}_1(F, j)$ gives the symbol of $T[\text{SA}[i] + 1] = T[\text{SA}[j]]$. The process continues until all $m$ symbols are extracted. Since the rank queries take constant time, the $\text{count}(P)$ operation on the CSA takes only $O(m \log n)$ time. The $\text{locate}(P)$ and $\text{display}(s, e)$ queries are answered using the same sampling techniques and backward search algorithm of the FM-Index. The RLCSA algorithm is based on the CSA [Mäkinen et al., 2009; Mäkinen et al., 2010].

Mäkinen et al. [2009]; Mäkinen et al. [2010] used RLCSA, RLFM+ and RLWT to compress two repetitive yeast collections, consisting of genomes from individual strains of each species. In general, the compression results are promising but the results achieved by standard compression algorithms are much better. However, this is expected, since the self-index needs to store extra information, such as the samples, to support the display, count and locate queries. From the three self-indexing algorithms, RLCSA is by far the best performer, in terms of the compressed size and the query performance. We compare the compression and random access performance of the index variants of our COMRAD and RLZ algorithms to the RLCSA index in Chapters 5 and 7, respectively. Next we discuss some LZ-based self-indexes.

### 3.4.3 Lempel-Ziv index

Lempel-Ziv indexes (LZ-Index) are built on text that has been parsed using either the LZ77 (Section 3.2.2) or the LZ78 (Section 3.2.2) algorithm. Recall that the LZ algorithms represent a string $T$ as a series of factors, where a factor is either a single literal or a reference to an earlier substring in $T$. For simplicity, let us define an LZ parsing of string $T$ into $d$ factors as a series of substrings $T = T_0 \ldots T_{d-1}$, where
each $T_i$ is the substring represented by the $i$th factor. If it was an LZ77 parsing, then each $T_i$ substring that is not a single literal, has occurred somewhere in the prefix $T_0 \ldots T_{i-1}$. If it was an LZ78 parsing, then for each substring $T_i$, the substring $T_i[0 \ldots |T_i| - 2]$ is one of the substrings in the set $\{T_0, \ldots, T_{i-1}\}$, and $T_i[|T_i| - 1]$ is a symbol in the alphabet $A$ of string $T$.

An index can be constructed on these factors to support the three queries $\text{count}(P)$, $\text{locate}(P)$ and $\text{display}(s, e)$. A $\text{display}(s, e)$ query can be handled trivially by an LZ-Index by using an extra data structure that specifies the start positions in $T$, where each factor $T_1 \ldots T_d$ starts. Once the factor that contains the position $s$ is determined, then recursively decoding each factor covering the range $T[s \ldots e]$ will extract the substring. Even though it seems that a $\text{display}(s, e)$ query can be answered in $O(e - s + 1)$ time, it is not the case, since more symbols than $e - s + 1$ symbols need to be extracted when recursively expanding factors. Below we describe an algorithm by Kreft and Navarro [2010] that allows a substring to be extracted in $O(e - s + 1)$ time for certain queries. In general, provided the initial factor can be found in constant time, and factors are stored such that a factor can be accessed in constant time, a $\text{display}(s, e)$ query can be answered in $O(e - s)$ time.

However, evaluating $\text{count}(P)$ and $\text{locate}(P)$ queries is non-trivial. Occurrences of a substring $P$ can either be contained completely within a factor or could span across two or more factors. We identify a substring that spans across two or more factors as a primary occurrence, and those contained completely within a factor as a secondary occurrence. Determining where substrings occur in an LZ-Index is difficult and requires extra data structures. Time and space trade-offs are made in the solutions that exist so far. Below we discuss the LZ-Index by Kärkkäinen and Ukkonen [1996]. The discussion is focused on the $\text{locate}(P)$ query, since $\text{count}(P)$ follows the same process as $\text{locate}(P)$.

**KU-LZI index** KU-LZI [Kärkkäinen and Ukkonen, 1996] uses an LZ77 parsing to obtain the set of factors. Two extra data structures are used to support $\text{locate}(P)$; a suffix tree $ST$ containing beginnings of the factors, and a reverse trie $RT$ storing each factor in reverse. The leaves of $ST$ contain the positions of occurrences in $T$ of the factor represented by the leaf, and the leaves of $RT$ contain the positions at which the reversed factor starts in $T$ (ie. where the forward direction factor ends).

To find primary occurrences of a substring $P$, first the substring is divided into two in all $m - 1$ possible ways, where $m = |P|$. Then for each partition $P[0 \ldots i]P[i \ldots m]$, where $0 \leq i \leq m - 2$, $P[0 \ldots i]$ is searched for in the reverse trie $RT$ to get the factors that end with $P[0 \ldots i]$ and $P[i \ldots m]$ is searched in $ST$ to get the factors that start with $P[i \ldots m]$. Searching in $ST$ and $RT$ gives positions at which the suffix $P[i \ldots m]$ ends in $T$ and the prefix $P[0 \ldots i]$ starts in $T$, respectively. Then a data structure supporting range checking is used to determine the pairs of factors that ends in $P[0 \ldots i]$ and starts with $P[i \ldots m]$. The positions at which these
factor pairs occur are the primary occurrences of $P$ in $T$.

The primary occurrences reveal the positions in $T$ where $P$ occurs. Therefore, secondary occurrences are obtained by identifying factors that completely contain each of the primary occurrences. Another array $S$ is required, which stores all the factors in sorted position order. Using $S$, we search for factors that completely contain each of the primary occurrences, and report the positions of occurrences.

Kärkkäinen and Ukkonen [1996] reports that the space usage of their data structure is only $O(n \log |A|)$ bits. The $count(P)$ and $locate(P)$ queries require $O(m^2 + m \log n + \frac{1}{\epsilon} occ \log^c n)$ time. Since the text $T$ is stored directly, $display(s,e)$ takes $O(e - s)$ time. KU-LZI is a theoretical result and was never implemented. Kreft and Navarro [2011] claims that the constant factors are too high to be practical.

Since KU-LZI, improvements have been made to the LZ-Index, mainly in terms of reducing the space usage. The indexes FM-LZI [Ferragina and Manzini, 2005] and NAV-LZI [Navarro, 2004], parses text $T$ using the LZ78 algorithm. Arroyuelo et al. [2012] provides a summary of the more recent LZ-Indexes and their performance. One of the latest LZ indexes is LZ-End, which is discussed next.

**LZ-End index** The LZ-End index [Kreft and Navarro, 2010, 2011] uses the LZ77 algorithm to parse the input string into factors, but with a slight modification to the parsing. A new factor must end at a position that a previous factor had ended. This ensures that substrings can be extracted in $O(e - s)$ time if position $e$ is at the end of a factor. A factor for substring $T[i...i + l + 1]$ is represented as a triplet $(q,l,c)$, where the substring is a suffix of $T_0...T_q$, $l$ is the length of the match, and $c$ is the last symbol of the substring $T[i+l]$. In the original LZ-End index that only supports $display(s,e)$, the $c$ symbols and the $q$ values of the factors are stored in separate arrays, and a bit vector stores the positions at which factors end in the original text $T$ so that the lengths can be inferred [Kreft and Navarro, 2010]. The complete LZ-End index stores the factors slightly differently, and also stores some extra data structures, such as a suffix trie of $T$ containing all the suffixes that are prefixes of factors, and a reverse suffix trie for all the reversed factors, among other data structures, to support the count and locate queries [Kreft and Navarro, 2011]. The same data structures can be used to create an index for a standard LZ77 parsing.

The LZ-End index can extract a substring $T[s...e]$ in $O(e - s)$ time, provided the substring ends at the end of a factor. If the substring ends at a factor $T_i = (q_i,l_i,c_i)$, then factor $T_{q_i} = (q_j,l_j,c_j)$ is recursively checked, and then factor $T_{q_j}$ is recursively checked, and so on. At each recursive lookup, symbol $c$ from the factor is extracted until all $e - s$ symbols are returned. Therefore, $e - s$ factors need to be accessed.

Extracting substrings that end at positions other than the end of a factor is more difficult, since more factors than the length of the substring need to be accessed.

The $locate(P)$ operation is implemented in a similar manner to that of KU-LZI. To find primary occurrences, the substring $P$ is partitioned in all $m - 1$ possible ways
into $P[0 \ldots i]P[i \ldots m]$, where $0 \leq i \leq m-2$. Then $P[i \ldots m]$ is searched in the suffix trie to find factors that begin with this substring, while $P[0 \ldots i]$ is searched in the reverse suffix trie to find factors that end with $P[0 \ldots i]$. Then the factors returned by these searches are filtered to get the occurrences that span multiple factors. The positions of these occurrences are then used to search for secondary occurrences.

The $\text{LZ-End}$ index requires $2d \log n + d \log d + d \log \delta + 5d \log |A| + O(d) + o(n)$ bits of storage, where $d$ is the number of factors, $n$ is the input length, $|A|$ is the input alphabet size, and $\delta$ is the maximum recursion depth of the factors. The index is asymptotically at most 2–3 times larger than the standard LZ77 compressed output [Kreft and Navarro, 2011]. Experimental results show that $\text{LZ-End}$ indexes are smaller than $\text{RLCSA}$ indexes. During index construction, memory that is 6–8 times the original input size is required, which is a significant overhead. For extracting text and for the locate query, $\text{LZ-End}$ index performs better than $\text{RLCSA}$. Therefore, $\text{LZ-End}$ is the most practical LZ77 index known to date.

In this section we introduced self-indexes, where a compressed dataset can be used as an index to efficiently extract substrings and to search for substrings, without decompressing the dataset. We described three types of indexes; those based on the Burrows-Wheeler Transform, compressed suffix arrays, and LZ-compressed data. Comparisons have been made on the effectiveness of $\text{FM-Index}$ based self-indexes and those based on the $\text{LZ-Index}$ [Sirén et al., 2009]. While LZ77 compression is good at capturing high-repetition compared to the $\text{FM-Index}$ or the other run-length compressed indexes, such as $\text{RLCSA}$, $\text{RLFM}$ or $\text{RLWT}$, not many practical $\text{LZ-Index}$ implementations existed [Kreft and Navarro, 2010; Sirén et al., 2009] until $\text{LZ-End}$.

Intuitively, an $\text{FM-Index}$ is better at answering $\text{count}(P)$ and $\text{locate}(P)$ queries, since these queries can be answered directly from using the suffix array. On the other hand, $\text{count}(P)$ and $\text{locate}(P)$ queries are more difficult to be answered by a $\text{LZ-Index}$, since the factors are not represented in a manner that supports searching. On the contrary, an $\text{LZ-Index}$ can answer $\text{display}(s,e)$ queries more easily than an $\text{FM-Index}$, which does not have direct access to the text as an $\text{LZ-Index}$.

Self-indexes produce promising compression results and are important solutions for storage and efficient access of large datasets. In Chapter 7, we extend our RLZ algorithm to construct a self-index, but with limitations that prevent the searching of substrings that span more than two factors. We also evaluate the performance of $\text{RLCSA}$ and $\text{LZ-End}$ against the $\text{COMRAD}$ and $\text{RLZ}$ indexes for compressing and accessing DNA sequences, in Chapters 5 and 7, respectively.

3.5 Chapter Summary

In this chapter we introduced various compression methods adopted by existing general-purpose and DNA-specific compression algorithms. The general-purpose compression algorithm discussion focused on dictionary compression algorithms,
which substituted repeated substrings with symbols that can be encoded using a smaller number of bits compared to the original substring. The discussion was separated into adaptive dictionary compression, where repeated substrings are detected dynamically in a single pass, and semi-static dictionary compression, where a dictionary of repeated substrings are determined prior to replacing repeated occurrences with references to entries in the dictionary. We hypothesise that semi-static algorithms will produce better compression results than adaptive algorithms, since more information is available to choose a better set of repeats to be encoded. We test this hypothesis with experiments in Chapter 4. Statistical compression and compression based on the Burrows-Wheeler transform were also discussed briefly. The discussion of general-purpose compression concluded with a brief introduction to encoding techniques required to complete the final encoding step of the compression process, and achieve the final compressed size.

Then, a comprehensive discussion of the existing DNA-specific compression algorithms was presented. DNA compression algorithms might be necessary because general-purpose algorithms are unaware of the repeat properties of DNA, such as reverse complement repeats and approximate repeats. Also, general-purpose compression algorithms are often implemented to detect local repeats but repeats in DNA are rarely local. The discussion of DNA compression algorithms was separated into algorithms that detected exact repeats, and those that detected exact and approximate repeats. Algorithms that detect exact repeats are generally fast, but are unable to detect the approximate repeats commonly present in DNA. Algorithms that detect approximate repeats on the other hand, achieve good compression results for DNA, but at higher resource costs in terms of compression time and memory usage. Nevertheless, some of the best compression results are achieved by algorithms that detect approximate repeats. In general, DNA compression algorithms tend to compress DNA sequences much better than general-purpose compression algorithms, as shown by many of the existing research in the area.

The final section briefly discussed some of the self-indexes that compress the input, and enables queries such as random access and search to be performed on the compressed input. The indexes are ideal for representing repetitive sequence collections, such as the ones from DNA sequencing projects, which need to be stored efficiently due to their large size, and also may need to be queried frequently.

So far we have only discussed the potential effectiveness of general-purpose and DNA-specific compression algorithms on compressing DNA datasets. In the next chapter, we use some of the publicly available implementations of the algorithms discussed in this chapter to compress various DNA sequences and sequence collections. Then we use the results to compare and contrast the algorithms in terms of their performance for compressing various test datasets.
Chapter 4

Existing compression algorithm evaluation

In the previous chapter, we introduced the basic concepts of compression and described some of the best-known and widely used general-purpose compression tools. We also stated that general-purpose compressors may be unsuitable for DNA sequence compression, and therefore DNA-specific methods might be necessary. In this chapter, we report experimental results for compressing test DNA datasets with these general-purpose compression tools. We compare the results in terms of the metrics described in Section 3.2.1, namely, compressed size, compression time, and memory usage during compression. We specifically compare the compressed size to the zero-order entropy, $H_0$, to show that some of these tools are unable to produce better results than that could be achieved with a simple encoding. We then use the same test datasets to evaluate publicly available DNA compression tools. A key observation made is the limitations on the size of the datasets that can be compressed using existing tools, greatly restricting their value for collection compression. We begin by describing the test data and the test environment used for the experiments.

4.1 Test data

The first dataset, discussed in Section 4.1.1, consists of larger complete genomes. Since single genomes tend to be non-repetitive, the genomes in this dataset may not give an indication of an algorithm’s ability to detect long range repetitions. Therefore, the second dataset, discussed in Section 4.1.2, contains a number of multi-sequence collections, and each collection contains sequences that either serve the same function or are genomes of distinct individuals of the same species. We did not use the standard test DNA sequences introduced by Grumbach and Tahi [1993], since the sequences are too small for demonstrating the scalability of algorithms for compressing large sequences. Below we discuss the contents of each dataset in detail and analyse their compressibility. The data sources are presented in Appendix A.
4.1.1 Single sequence dataset

This dataset contains sequences to test single-sequence compression performance. The sequence sizes are comparable to the types of sequences that are typically available online, such as whole chromosomes and genomes. The dataset is named \texttt{DATASET-sin}, and it contains the following eight genomes: PfalcRef, the reference genome of the malaria-causing virus \textit{Plasmodium falciparum} (\textit{P. falciparum}), CelegRef, the reference genome of the roundworm \textit{Caenorhabditis elegans} (\textit{C. elegans}), AthalRef, the reference genome of the mustard-like plant \textit{Arabidopsis thaliana} (\textit{A. thaliana}), EcoliK12, the genome of the laboratory strain K-12 of the intestinal bacteria \textit{Escherichia coli} (\textit{E. coli}), DmelRef, the reference genome of the fruit fly \textit{Drosophila melanogaster} (\textit{D. melanogaster}), GgalRef, the chicken reference genome \textit{Gallus gallus} (\textit{G. gallus}), ScereRef, reference genome of the yeast species \textit{Saccharomyces cerevisiae} (\textit{S. cerevisiae}), and HsapRef, the reference human genome \textit{Homo sapiens} (\textit{H. sapiens}). For each genome, the individual chromosome sequences are concatenated to form a single sequence. The genomes range in size from a few megabases to around 3 Gbases for the \textit{H. sapiens} genome. Table 4.1a contains the genomes and their sizes.

4.1.2 Repetitive dataset

In this thesis we explore algorithms for compressing highly repetitive collections. These collections contain many individual sequences from the same species or many sequences that have the same functionality, such that the sequences are identical except for a small amount of variation that occurs as a result of mutations present in the individual sequences. These are the kinds of collections that are available from various sequencing projects as described in Section 2.2.3. Due to the high level of repetition, the compression from these collections is expected to be much greater than that for single-sequence compression. This dataset is named \texttt{DATASET-rep}.

The first three collections in \texttt{DATASET-rep} contain DNA sequences from genes that serve the same or similar functions. The \texttt{Hemo} collection contains 15,199 DNA sequences that encode proteins associated with \textit{Hemoglobin} from various species. \textit{Hemoglobin} is the protein that transports oxygen in the red blood cells of vertebrates. The collection named \texttt{Infl} contains 78,041 gene sequences for various strains of the \textit{Influenza} virus. The \texttt{Mito} collection contains 1,521 \textit{Mitochondria} genomes from various species. \textit{Mitochondria} is an organelle in eukaryotic cells, whose main function is to convert the chemical energy from food into the form that cells can use to perform its functions. The genome largely consists of coding regions and contains little repetition but, given the high proportion of coding regions, we can expect that there is conservation between the genomes of the different species.

The remaining collections consist of multiple complete genomes from the same species. The \texttt{Athal} collection contains contigs for four different strains of the
4.1. TEST DATA

*A. thaliana* plant from the 1001 Genomes project, and the reference genome for the species. The *Scere* and *Spara* collections contain genomes of 39 and 36 strains from two yeast species, *S. cerevisiae* and *S. paradoxa* (*Saccharomyces paradoxa*), respectively. The *Ecol* collection contains the genomes of 33 strains of the *E. coli* bacteria. The *Sson* collection contains assembled contigs for the genomes of 230 individuals from the *S. sonnei* (*Shigella sonnei*) bacteria and the fully assembled reference *S. sonnei* genome. The *Hsap* collection is the largest, and consists of the four publicly available assembled human genomes, which are the reference human genome (NCBI build 37 release on 2nd March, 2009), the Craig Venter genome [Levy *et al.*, 2007], and the genomes of the Han Chinese [Wang *et al.*, 2008] and Korean individuals [Ahn *et al.*, 2009].

To show the results of compressing a set of multiple sequences that share less material than the remaining collections, we also add a collection of bacterial genomes, identified as *Bact*. Unlike the other, this collection consists of 1,446 genomes from various species of *Bacteria*, rather than genomes from the same species.

The collections and their sizes are listed in Table 4.1b. Most tools used for our experiments are not designed for detecting repeats across multiple sequences. Separately compressing individual sequences in a collection will not sufficiently compress the collection due to the low level of repetitions present within sequences. Therefore, to detect the repeats present in the collections in dataset-rep, we concatenate the individual sequences in a collection to create a single sequence. We do not add any extra symbols to indicate sequence boundaries in the concatenated sequences, because most DNA-specific compression tools only accept sequences with symbols from the standard DNA alphabet, namely the symbols a, c, g and t. This type of preprocessing is undesirable as individual sequences can no longer be accessed as in the uncompressed collection. Preprocessing large collections is also impractical.

The above test datasets contain DNA sequences with nucleotides from the extended DNA alphabet. Since some DNA compressors only accept sequences containing nucleotides from the standard DNA alphabet, for the experiments in this chapter, we substitute all nucleotides not in the standard DNA alphabet with the nucleotide a. Most of the substitutions are for consecutive n nucleotides and substituting a run of ns with a specific nucleotide will result in the repetitiveness of the sequences being altered as minimally as possible. Although substituting each n with a random symbol from $\sigma$ would be more consistent with the definition of n, we do not do so to preserve the information content of the sequences. Even though the general-purpose compression tools can compress sequences containing nucleotides from the extended DNA alphabet, we run all experiments with the altered datasets to consistently compare the general-purpose tools to the DNA compression tools.
### 4.1.3 Dataset repeat properties

We conduct a simple analysis of the repeat properties of the sequences in the two test datasets. For each sequence in each dataset, we calculate the total number of $k$-mers, the total number of distinct $k$-mers and the frequencies of each distinct $k$-mer. Using this information, we calculate some statistics to estimate the level of repetition in each sequence.

First we calculate the number of times on average a $k$-mer is repeated (total $k$-mers divided by distinct $k$-mers). The higher the average repeat rate, the higher the level of repetition in the sequence, since each distinct $k$-mer is likely to be repeated many times. We also calculate the percentage of distinct $k$-mers with a frequency of at least 2 relative to the total number of $k$-mers. If there are repetitions in the dataset, this percentage indicates whether many distinct $k$-mers are repeated a few times in the dataset or only a few distinct $k$-mers are repeated many times.

For this experiment we chose $k = 15$. For a randomly generated string from the standard DNA alphabet $\sigma$, there is a $1$ in $|\sigma|^k$ probability that the given $k$-mer will appear in the string. With $k = 15$, there is a $1$ in $2^{30}$ probability that any given 15-mer will appear in a sequence. Therefore, the likelihood of any given 15-mer occurring in a sequence more than once is low unless the 15-mer is a repeat.

Table 4.2 contains the repeat analysis results for the two datasets. The sequences in DATASET-SIN have a low level of repetition, with a majority of sequences containing mostly distinct 15-mers. The GgalRef and HsapRef sequences have the highest level of repeats, with half to two-thirds of distinct 15-mers being repeated at least twice.

The multi-sequence collections in DATASET-REP are more repetitive than the sequences in DATASET-SIN. Despite our expectations of sequence similarity, the Mito collection is the least repetitive, with a 15-mer being repeated twice on average.
### 4.1. TEST DATA

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Total 15-mers</th>
<th>Distinct 15-mers</th>
<th>Avg. Repeat Rate</th>
<th>Distinct 15-mers F ≥ 2</th>
<th>Percent 15-mers F ≥ 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>4,639,675</td>
<td>4,517,621</td>
<td>1.03</td>
<td>74,303</td>
<td>1.60%</td>
</tr>
<tr>
<td>ScereRef</td>
<td>12,162,996</td>
<td>11,301,518</td>
<td>1.08</td>
<td>413,245</td>
<td>3.40%</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>23,264,338</td>
<td>12,516,761</td>
<td>1.86</td>
<td>2,631,854</td>
<td>11.31%</td>
</tr>
<tr>
<td>CellegRef</td>
<td>100,272,210</td>
<td>69,674,169</td>
<td>1.44</td>
<td>12,224,517</td>
<td>12.19%</td>
</tr>
<tr>
<td>AthalRef</td>
<td>119,146,348</td>
<td>87,734,526</td>
<td>1.36</td>
<td>16,221,376</td>
<td>13.61%</td>
</tr>
<tr>
<td>DmelRef</td>
<td>168,736,537</td>
<td>102,192,724</td>
<td>1.65</td>
<td>16,290,185</td>
<td>9.65%</td>
</tr>
<tr>
<td>GgalRef</td>
<td>1,031,883,471</td>
<td>388,909,323</td>
<td>2.65</td>
<td>205,782,420</td>
<td>19.94%</td>
</tr>
<tr>
<td>HsapRef</td>
<td>3,095,677,412</td>
<td>483,820,305</td>
<td>6.40</td>
<td>301,346,610</td>
<td>9.73%</td>
</tr>
</tbody>
</table>

(a) **dataset-sin**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>7,381,028</td>
<td>2,005,322</td>
<td>3.68</td>
<td>389,725</td>
<td>5.28%</td>
</tr>
<tr>
<td>Mito</td>
<td>25,260,398</td>
<td>12,885,123</td>
<td>1.96</td>
<td>3,267,684</td>
<td>12.94%</td>
</tr>
<tr>
<td>Infl</td>
<td>112,640,907</td>
<td>2,185,912</td>
<td>51.53</td>
<td>1,339,207</td>
<td>1.19%</td>
</tr>
<tr>
<td>Ecol</td>
<td>164,898,725</td>
<td>22,156,195</td>
<td>7.44</td>
<td>15,640,059</td>
<td>9.48%</td>
</tr>
<tr>
<td>Athal</td>
<td>506,663,306</td>
<td>99,850,690</td>
<td>5.07</td>
<td>83,915,295</td>
<td>16.56%</td>
</tr>
<tr>
<td>Scere</td>
<td>485,873,138</td>
<td>15,142,131</td>
<td>32.09</td>
<td>14,067,324</td>
<td>2.90%</td>
</tr>
<tr>
<td>Spara</td>
<td>429,265,788</td>
<td>20,423,428</td>
<td>21.02</td>
<td>17,679,897</td>
<td>4.12%</td>
</tr>
<tr>
<td>Sson</td>
<td>966,455,412</td>
<td>17,491,841</td>
<td>55.25</td>
<td>12,328,942</td>
<td>1.28%</td>
</tr>
<tr>
<td>Bact</td>
<td>2,770,539,258</td>
<td>631,545,169</td>
<td>4.39</td>
<td>375,217,093</td>
<td>13.54%</td>
</tr>
<tr>
<td>Hsap</td>
<td>12,066,063,707</td>
<td>475,421,591</td>
<td>25.38</td>
<td>442,929,822</td>
<td>3.67%</td>
</tr>
</tbody>
</table>

(b) **dataset-rep**

Table 4.2: The repeat properties of the sequences in **dataset-sin** and **dataset-rep**. The columns are: sequence or collection name, total number of 15-mers, total number of distinct 15-mers, average repeat rate of each distinct 15-mer, total number of distinct 15-mers with a frequency of at least 2, and the number of distinct 15-mers with frequency of at least 2 as a percentage of the total 15-mers, respectively.

The **Infl** and **Sson** collections have the highest level of repetition, with a 15-mer occurring over 50 times on average.

The repeat analysis in this section is only a guide to the repetitiveness of the sequences in each dataset rather than being an accurate analysis. For example, this analysis does not cover any repeats of length less than 15, or any approximate repeats, which are common in DNA sequences. Most compression tools are able to detect repeats of various lengths. However, as we will see shortly, the compression results for the general-purpose tools reflect the results of this repeat analysis. Next, we describe the testing methodology used to analyse the performance of each tool.
4.2 Testing methodology

We compressed each dataset using a selection of general-purpose and DNA compression algorithms whose implementations are publicly available. We first analyse the performance of the general-purpose algorithms, followed by the DNA compression algorithms. Then we compare the results of the two types of algorithms to determine whether DNA compression algorithms are necessary for compressing DNA. In this section, we describe the performance metrics used to compare the algorithms. All experiments were run on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single processor.

4.2.1 Measuring performance

In Section 3.2.1, we described three criteria for measuring the performance of compression tools, namely, compressed size, compression and decompression speed and memory usage. Below we discuss the measurements obtained for each criteria.

Compressed size

The compressed size of a sequence or collection of sequences indicates the amount of space that has been saved by using a compression tool. The original size of a sequence can be measured in terms of the number of bases in the sequence, although as discussed in Section 3.3, this is not an accurate representation of the amount of space required to store a sequence on disk. It is straightforward to use either 2 bpb (bits per base) or 4 bpb for the standard DNA alphabet or the extended DNA alphabet, respectively, to store sequences. It is also possible to use a simple encoding such as Huffman or arithmetic encoding to store a sequence in space that is close to $nH_0$ bits, where $n$ is the number of bases and $H_0$ is the zero-order entropy of the sequence. In practice $H_0$ is very close to 2 bpb even for a sequence derived from the extended alphabet, since the symbols other than a, c, g, t and n are extremely rare.

Since our test sequences only contain nucleotides from the standard alphabet, the smallest uncompressed representation of a DNA sequence can be achieved with a 2 bpb encoding, which we identify as the naïve encoding. We consider a sequence to be compressed if a compression tool can compress the sequence to a smaller size than using a naïve encoding. To measure the compressed size of a sequence, we calculate the total number of bytes on disk required to store all the information necessary for the decompression tool to return the original sequence.

We also compare the average number of bits required by each tool to encode a base, to the 0-order entropy, $H_0$. If a compression tool requires a lower average number of bits than $H_0$ to encode a base, then enough repetition is present in the sequence to store it in less space than using a standard encoder. In summary, the lower the compressed size, the better the compression performance of the tool.
4.2. TESTING METHODOLOGY

Compression and decompression time

The time taken to compress is a measure of the efficiency of a compression tool. For certain applications, fast compression speeds may be important, while other applications may require the data to be decompressed more frequently than it is compressed. In the latter case, decompression speed is much more significant than the compression speed. In general, fast compression and decompression speeds are favourable. We calculate the compression and decompression time as the total time taken to read data from disk, to process the data, and to write the compressed or decompressed data to disk. The time is measured in seconds using the real time reported by the Unix `time` utility.

Memory usage

Memory usage is another measure of the efficiency of a compression tool. Generally, the lower the memory used during compression or decompression, the greater the efficiency of the tool. Certain compression methods are memory intensive, so much so that they cannot be applied to large volumes of data. To permit their use across a wide variety of machines, general-purpose tools carefully limit the memory resources used, often allowing the user to specify these limits.

Maximum memory use was measured only during compression but not during decompression, since the memory used during decompression tends to be less significant. We measured the approximate maximum memory used by compression tools using the `massif` tool of the Unix utility `valgrind`. We enabled the `--pages-as-heap` option so that the memory use in terms of page allocations, which not only includes the heap memory allocated, but also includes the memory used by the code, data and bss sections of the process, as well as the stack memory used, is measured. `massif` outputs snapshots of the peak memory used at various points during the execution of a process, so we parsed this output to retrieve the snapshot containing the largest memory use value. We report this value as the approximate maximum memory used by a compression tool, measured in megabytes.

Unfortunately, `massif` can only be used for tools that are implemented in C or C++, so the memory usage of two DNA compression tools could not be measured with it. One was `GenCompress`, for which we only had a 32-bit binary, and our version of `massif` was compiled for a 64-bit environment. The other is `XMCompress`, which is a Java executable. To estimate the memory use of these two tools, a script was used that records the resident set size of the process every millisecond. The maximum resident set size value was reported as an estimate for the approximate maximum memory used by these two tools. We acknowledge that this estimate is not as accurate as those given by `massif`, but the values are a good estimate of the relative memory usage of the different tools.
4.3 General-purpose compression algorithm evaluation

Having established an evaluation framework, in this section we analyse the performance of the general-purpose compression algorithms discussed in Section 3.2 for compressing DNA sequences. We measure the performance of each implementation based on the criteria of compressed size, compression and decompression speed, and approximate maximum memory use as described in Sections 3.2.1 and 4.2.1.

The following set of publicly available implementations are used for the experiments. In the category of adaptive dictionary compression, gzip and 7-Zip are used for LZ77 compression, compress for LZ78 compression, and Sequitur for global repeat detection. For semi-static dictionary compression, Re-pair is used. Finally, to test statistical compression and Burrows-Wheeler transform based compression, ppmd and bzip2 are used, respectively. The sources of the compression implementations are presented in Appendix A.

4.3.1 Input parameters

In Chapter 3 we hypothesised that general-purpose compression tools will not produce good compression results for DNA sequences because they are not tuned to detect the types of repeats commonly observed in DNA. To allow these tools to produce the best possible results for DNA sequences, we set the parameters to favour better compression at the cost of speed. The parameters used are as follows:

gzip: The amount of memory to be used by the dictionary cannot be specified in gzip. Instead, a parameter that sets the desired level of compression can be set. A parameter value between 1–9 can be chosen, where 1 has the fastest compression speed but likely to achieve less compression, while 9 has the slowest speed but achieve the best compression result. We favour better compression results over fast compression so we chose the 9 option.

7-Zip: We use 7-Zip with the LZMA encoder. As for gzip we chose the value 9 for the level of compression parameter to favour compression over speed. In 7-Zip, the dictionary size can be specified and we set it to the maximum value of 1 Gbyte to achieve the best compression possible.

compress: Only the parameter for the maximum number of bits to be used to encode dictionary entries can be set in compress. The tool begins by using 8 bit codewords, and once the 8-bit codewords are consumed, it uses 9-bit codewords, then 10-bit codewords and so on until the specified maximum is reached. The higher the maximum limit, the more entries that can be included in the dictionary and the better the compression. Therefore, we set this parameter to the maximum permitted value of 16.
Sequitur: For Sequitur, we set the maximum memory use limit for the hash table to the maximum permitted value of 2 Gbytes. Other parameters can be set to limit the number of rules and to increase the frequency of symbol pairs that must occur before being substituted by a non-terminal. These parameters exist to limit the memory use at a cost to the compressed size so we do not place any such limits. By default, symbol pairs must occur at least twice before being substituted by a non-terminal and there are no limits on the number of rules. However, for some of the larger sequences exceeding 1 Gbase, Sequitur requires more memory than is available on our platform. In these cases, instead of setting the above parameters, we chose to compress the sequences in equal-sized blocks. In the results section we indicate which sequences were compressed in this manner.

Re-pair: Many parameters can be set to lower the memory use of Re-pair and to increase the compression speed. These include the number of times a symbol pair must occur before being substituted by a non-terminal; whether to compress the input in multiple blocks; the maximum number of substitution rules to be introduced; and the maximum length of the right hand side of a rule. For most sequences, we do not place any limitations. For large sequences exceeding 1 Gbase, Re-pair requires more memory than available on our platform. In these cases, we set the block size parameter to compress the sequences in equal-sized blocks.

ppmd: We were unable to find a 64-bit version of the ppmd implementation for our test machine, so we used the ppmd encoder of 7-Zip. We set the compression level to 9, memory size to the 2 Gbytes maximum limit and the model order to the maximum value of 32 to achieve the best compression possible at the cost of compression speed.

bzip2: As for gzip2 and 7-Zip, we set the compression level parameter of bzip2 to 9 to achieve the best compression possible.

4.3.2 Results

We now analyse the performance of the general-purpose compression tools for compressing the sequences in dataset-sin and dataset-rep. Sequitur and Re-pair could not compress some larger sequences due to insufficient memory. Such sequences were compressed in equal-sized blocks, where each block was compressed independently. In the results, these results are distinguished by a * symbol.

Compressed size

The compressed sizes, measured in megabytes, are reported in Table 4.3. The results can be compared to the size achieved by the 2 bpb representation (second column). As shown in Table 4.2a, EcoliK12 and ScereRef are the only non-repetitive
sequences in \textsc{dataset-sin}, so we expected none of the tools to compress these sequences to a size smaller than the 2 bpb representation. This assumption was mostly confirmed by the results, except for \textit{7-Zip} and \textit{Sequitur}, which were able to compress the two sequences. We expected the remaining sequences to be compressed.

For single-sequence compression and for compressing the \textit{Mito} and \textit{Bact} collections in \textsc{dataset-rep}, \textit{Re-pair} produced the worst results, as it is unsuitable for compressing non-repetitive or moderately repetitive sequences. However, \textit{Re-pair} performed well on the remaining collections in \textsc{dataset-rep}, showing that \textit{Re-pair} is only suitable for compressing highly repetitive sequences.

The \textit{gzip} tool produced better results than \textit{Re-pair} for the less repetitive sequences in the two test datasets, but was mostly unable to compress the sequences to a size smaller than the 2 bpb representation. Especially for \textsc{dataset-rep}, most of the time, \textit{gzip} produced larger sizes than the other tools. This is due to the repeats in these collections being further apart than could be detected by the \textit{gzip} window size. This is especially evident in the poor results for the \textit{Sson}, \textit{Spara} and \textit{Scere} collections, which were all highly repetitive. On the other hand, \textit{gzip} was able to compress the smaller repetitive collections, \textit{Hemo} and \textit{Infl}.

The \textit{bzip2} and \textit{compress} tools produced slightly better results than \textit{gzip}. For both test datasets, these tools could not produce a smaller size than the 2 bpb representation for most sequences, except for the two or three larger sequences or collections. This shows that neither \textit{gzip}, \textit{compress} nor \textit{bzip2} are suitable for compressing large repetitive DNA sequence collections.

Both \textit{ppmd} and \textit{Sequitur} compressed well and out-performed the 2 bpb representation for \textsc{dataset-sin}, although \textit{Sequitur} performed marginally better than \textit{ppmd}. For \textsc{dataset-rep}, \textit{ppmd} performed well on the three smallest collections, but not as well on the larger collections. For the moderately repetitive sequences, \textit{Sequitur} performed well, while \textit{Re-pair} performed well on highly repetitive sequences.

The best compression performance in terms of the compressed size was achieved by \textit{7-Zip} for both test datasets. As well as effectively compressing single genomes from \textsc{dataset-sin}, it was also able to compress the non-repetitive collections, \textit{Mito} and \textit{Bact}. The most significant result is its ability to compress the \textit{Hsap} collection of four genomes to just 655 Mbytes.

Overall, \textit{Sequitur}, \textit{ppmd} and \textit{7-Zip} achieved the best compressed sizes for single genomes. It is interesting to note that, for larger sequences, \textit{ppmd} no longer performs well, since it is too expensive to keep the contexts for all of the input seen so far. On the other hand, the larger window size enables \textit{Sequitur} and \textit{7-Zip} to consider all or most of the input seen so far to achieve better compression for larger sequences. For the repetitive sequence collections, all tools except for \textit{gzip}, \textit{compress} and \textit{bzip2}, performed well, with \textit{7-Zip} out-performing most other tools by significant margins. These results are further confirmed by Table 4.4, where the average bits required by each to encode a nucleotide is less than the 0-order entropy for most tools.
Table 4.3: Compressed sizes in Mbytes produced by general-purpose compressors. The second column contains the results for a 2 bpb representation, and the remaining columns contain results for each tool. In dataset-sin, the HsapRef sequence was divided into two equal-sized blocks to compress with Sequitur, and GgalRef and HsapRef sequences were divided into two and four equal-sized blocks, respectively, to compress with Re-pair. In dataset-rep, for Sequitur and Re-pair, the Bact collection was compressed in 4 equal-sized blocks, and the Hsap collection was compressed in 24 blocks, where each block contained the four copies of each chromosome.
CHAPTER 4. EXISTING COMPRESSION ALGORITHM EVALUATION

### Table 4.4: The average number of bits used by the general-purpose compressors to encode a base. The second column contains the 0-order entropy of each sequence and the remaining columns contain the results for each tool. The entries with the ∗ symbols were compressed in blocks. Refer to Table 4.3 for more information.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>0-ord.</th>
<th>gzip</th>
<th>7-Zip</th>
<th>compress</th>
<th>Seq.</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>2.00</td>
<td>2.24</td>
<td>2.04</td>
<td>2.10</td>
<td>2.07</td>
<td>4.61</td>
<td>2.06</td>
<td>2.16</td>
</tr>
<tr>
<td>ScereRef</td>
<td>1.96</td>
<td>2.23</td>
<td>1.98</td>
<td>2.09</td>
<td>2.00</td>
<td>4.19</td>
<td>2.02</td>
<td>2.17</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>1.71</td>
<td>1.91</td>
<td>1.65</td>
<td>1.79</td>
<td>1.68</td>
<td>3.35</td>
<td>1.68</td>
<td>1.91</td>
</tr>
<tr>
<td>CelegRef</td>
<td>1.94</td>
<td>2.15</td>
<td>1.86</td>
<td>2.04</td>
<td>1.91</td>
<td>3.54</td>
<td>1.93</td>
<td>2.09</td>
</tr>
<tr>
<td>AthalRef</td>
<td>1.94</td>
<td>2.17</td>
<td>1.81</td>
<td>2.06</td>
<td>1.94</td>
<td>3.55</td>
<td>1.96</td>
<td>2.12</td>
</tr>
<tr>
<td>DmelRef</td>
<td>1.97</td>
<td>2.05</td>
<td>1.52</td>
<td>2.05</td>
<td>1.63</td>
<td>2.88</td>
<td>1.59</td>
<td>1.96</td>
</tr>
<tr>
<td>GgalRef</td>
<td>1.96</td>
<td>2.08</td>
<td>1.84</td>
<td>1.97</td>
<td>1.86</td>
<td>3.23*</td>
<td>1.91</td>
<td>2.04</td>
</tr>
<tr>
<td>HsapRef</td>
<td>1.95</td>
<td>1.99</td>
<td>1.61</td>
<td>1.94</td>
<td>1.70*</td>
<td>2.86*</td>
<td>1.72</td>
<td>1.90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coll.</th>
<th>0-ord.</th>
<th>gzip</th>
<th>7-Zip</th>
<th>compress</th>
<th>Seq.</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>2.00</td>
<td>0.87</td>
<td>0.67</td>
<td>1.57</td>
<td>0.98</td>
<td>1.64</td>
<td>0.68</td>
<td>0.86</td>
</tr>
<tr>
<td>Mito</td>
<td>1.95</td>
<td>2.00</td>
<td>1.05</td>
<td>2.05</td>
<td>1.70</td>
<td>2.98</td>
<td>1.35</td>
<td>1.90</td>
</tr>
<tr>
<td>Infl</td>
<td>1.97</td>
<td>0.56</td>
<td>0.11</td>
<td>1.85</td>
<td>0.36</td>
<td>0.29</td>
<td>0.13</td>
<td>0.54</td>
</tr>
<tr>
<td>Ecol</td>
<td>2.00</td>
<td>2.24</td>
<td>0.27</td>
<td>2.10</td>
<td>0.55</td>
<td>0.58</td>
<td>0.69</td>
<td>2.16</td>
</tr>
<tr>
<td>Spara</td>
<td>1.95</td>
<td>2.16</td>
<td>0.10</td>
<td>2.01</td>
<td>0.23</td>
<td>0.22</td>
<td>1.24</td>
<td>2.09</td>
</tr>
<tr>
<td>Scere</td>
<td>1.93</td>
<td>2.10</td>
<td>0.08</td>
<td>1.94</td>
<td>0.17</td>
<td>0.14</td>
<td>1.18</td>
<td>2.02</td>
</tr>
<tr>
<td>Athal</td>
<td>1.93</td>
<td>2.12</td>
<td>0.52</td>
<td>1.97</td>
<td>0.73</td>
<td>0.85</td>
<td>1.94</td>
<td>2.06</td>
</tr>
<tr>
<td>Sson</td>
<td>2.00</td>
<td>2.22</td>
<td>0.07</td>
<td>2.07</td>
<td>0.14</td>
<td>0.12</td>
<td>0.70</td>
<td>2.14</td>
</tr>
<tr>
<td>Bact</td>
<td>2.00</td>
<td>2.19</td>
<td>1.43</td>
<td>2.12</td>
<td>1.78*</td>
<td>2.93*</td>
<td>1.96</td>
<td>2.12</td>
</tr>
<tr>
<td>Hsap</td>
<td>1.96</td>
<td>2.03</td>
<td>0.43</td>
<td>1.95</td>
<td>0.60*</td>
<td>0.60*</td>
<td>1.74</td>
<td>1.94</td>
</tr>
</tbody>
</table>

(b) DATASET-REP

Compression and decompression speed

Next we compare the performance of the compression tools in terms of the time taken to compress and decompress. Table 4.5 lists the compression times in seconds for each tool. In general, 7-Zip tends to be the slowest for smaller sequences in both test datasets. However, for the two largest sequences in DATASET-SIN, and the Athal and Bact collections in DATASET-REP, Re-pair was the slowest, and this is due to compressing the sequences in blocks. For both test datasets, the compress tool was consistently faster than the other tools, and only took 9 minutes to compress the four human genomes. The bzip2 tool was also fast but slower than compress. The tools, Sequitur, gzip and ppmd, had moderately fast compression speeds, with ppmd being the fastest of the three. The results for DATASET-SIN show that Sequitur and ppmd have a good trade-off between compression speed and compression results for single-sequence compression. For DATASET-REP, of the three compressors that produced good results, Sequitur was the fastest, followed by 7-Zip and Re-pair. Overall, if good compression is more important than the compression speed, then 7-Zip is more suitable for both single-sequence and collection compression.
Table 4.5: The time taken to compress in seconds for the general-purpose compressors. The entries with * symbols were compressed in blocks. Refer to Table 4.3 for more information.

The decompression times for each tool are presented in Table 4.6. All dictionary compressors, except Sequitur, were fast to decompress. However Re-pair took a relatively long time to decompress the Hsap collection, since 24 separate blocks needed to be decompressed. The decompression speed of Sequitur could not be analysed for large sequences due to a software error. For small single sequences, Sequitur was faster to decompress than ppmd but much slower than the other tools. The bzip2 tool was slower to decompress than the LZ implementations, but ppmd was the slowest, with its decompression speed being slower than its compression speed, as expected from a statistical compressor. Therefore, if fast decompression is important, then ppmd should not be used, whereas the other tools are able to decompress much faster than their respective compression speeds.

Memory usage

Finally, we analyse the performance of the general-purpose compression tools in terms of the maximum memory used during compression, and the results are reported in Table 4.7. The gzip tool, followed by compress and bzip2, used the least memory and the memory use was independent of the sequences being compressed.
CHAPTER 4. EXISTING COMPRESSION ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>Dataset</th>
<th>gzip</th>
<th>7-Zip</th>
<th>compress</th>
<th>Sequitur</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>≪1</td>
<td>≪1</td>
<td>≪1</td>
<td>3</td>
<td>≪1</td>
<td>2</td>
<td>≪1</td>
</tr>
<tr>
<td>ScereRef</td>
<td>≪1</td>
<td>≪1</td>
<td>≪1</td>
<td>4</td>
<td>≪1</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>≪1</td>
<td>≪1</td>
<td>≪1</td>
<td>6</td>
<td>≪1</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>CellegRef</td>
<td>1</td>
<td>3</td>
<td>≪1</td>
<td>28</td>
<td>2</td>
<td>91</td>
<td>10</td>
</tr>
<tr>
<td>AthalRef</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>—</td>
<td>3</td>
<td>95</td>
<td>12</td>
</tr>
<tr>
<td>DmelRef</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>—</td>
<td>5</td>
<td>114</td>
<td>16</td>
</tr>
<tr>
<td>GgalRef</td>
<td>10</td>
<td>34</td>
<td>9</td>
<td>—</td>
<td>30*</td>
<td>1,040</td>
<td>125</td>
</tr>
<tr>
<td>HsapRef</td>
<td>33</td>
<td>98</td>
<td>29</td>
<td>—</td>
<td>92*</td>
<td>2,750</td>
<td>328</td>
</tr>
</tbody>
</table>

(a) DATASET-SIN

<table>
<thead>
<tr>
<th>Coll.</th>
<th>gzip</th>
<th>7-Zip</th>
<th>compress</th>
<th>Sequitur</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>≪1</td>
<td>≪1</td>
<td>≪1</td>
<td>3</td>
<td>≪1</td>
<td>2</td>
<td>≪1</td>
</tr>
<tr>
<td>Mito</td>
<td>≪1</td>
<td>≪1</td>
<td>≪1</td>
<td>8</td>
<td>≪1</td>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>Infl</td>
<td>≪1</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td>1</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>Ecol</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>—</td>
<td>6</td>
<td>72</td>
<td>17</td>
</tr>
<tr>
<td>Spara</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>—</td>
<td>19</td>
<td>306</td>
<td>45</td>
</tr>
<tr>
<td>Scere</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>—</td>
<td>20</td>
<td>337</td>
<td>49</td>
</tr>
<tr>
<td>Athal</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>—</td>
<td>30</td>
<td>425</td>
<td>64</td>
</tr>
<tr>
<td>Sson</td>
<td>10</td>
<td>14</td>
<td>8</td>
<td>—</td>
<td>42</td>
<td>485</td>
<td>100</td>
</tr>
<tr>
<td>Bact</td>
<td>36</td>
<td>82</td>
<td>30</td>
<td>—</td>
<td>92*</td>
<td>2,279</td>
<td>297</td>
</tr>
<tr>
<td>Hsap</td>
<td>425</td>
<td>456</td>
<td>319</td>
<td>—</td>
<td>1,083*</td>
<td>12,321</td>
<td>1,437</td>
</tr>
</tbody>
</table>

(b) DATASET-REP

Table 4.6: The time taken to decompress in seconds for the general-purpose compressors. The entries with * symbols were compressed in blocks. Refer to Table 4.3 for more information. For Re-pair, the Hsap collection was decompressed in blocks. The Bact sequence was also decompressed in blocks but the blocks were handled by the tool so the impact on the decompression cost was lower. The — symbols indicate that a tool was unable to decompress a collection.

The test dataset DATASET-REP is the most interesting, given that it contains the types of DNA sequence collections that will be commonly available through DNA sequencing projects. As predicted, general purpose compression tools like gzip,
4.3. GENERAL-PURPOSE COMPRESSION ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>Dataset</th>
<th>gzip</th>
<th>7-Zip</th>
<th>comp.</th>
<th>Sequitur</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>6.05</td>
<td>121.45</td>
<td>7.01</td>
<td>1,949.71</td>
<td>152.52</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>ScereRef</td>
<td>6.05</td>
<td>194.45</td>
<td>7.01</td>
<td>1,990.71</td>
<td>375.43</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>6.05</td>
<td>340.45</td>
<td>7.01</td>
<td>2,025.71</td>
<td>704.47</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>CelegRef</td>
<td>6.05</td>
<td>1,088.45</td>
<td>7.01</td>
<td>2,375.71</td>
<td>2,981.99</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>AthalRef</td>
<td>6.05</td>
<td>1,392.45</td>
<td>7.01</td>
<td>2,483.71</td>
<td>3,523.74</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>DmelRef</td>
<td>6.05</td>
<td>2,128.45</td>
<td>7.01</td>
<td>2,584.71</td>
<td>4,903.78</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>GgalRef</td>
<td>6.05</td>
<td>10,800.45</td>
<td>7.01</td>
<td>5,840.71</td>
<td>15,067.18 ∗</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>HsapRef</td>
<td>6.05</td>
<td>10,800.45</td>
<td>7.01</td>
<td>10,093.70 ∗</td>
<td>20,039.20 ∗</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
</tbody>
</table>

(a) dataset-sin

<table>
<thead>
<tr>
<th>Coll.</th>
<th>gzip</th>
<th>7-Zip</th>
<th>comp.</th>
<th>Sequitur</th>
<th>Re-pair</th>
<th>ppmd</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>6.05</td>
<td>140.45</td>
<td>7.01</td>
<td>1,946.71</td>
<td>224.32</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Mito</td>
<td>6.05</td>
<td>416.45</td>
<td>7.01</td>
<td>2,044.71</td>
<td>764.80</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Inf1</td>
<td>6.05</td>
<td>1,392.45</td>
<td>7.01</td>
<td>2,053.71</td>
<td>3,173.92</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Ecol</td>
<td>6.05</td>
<td>2,128.45</td>
<td>7.01</td>
<td>2,214.71</td>
<td>4,850.51</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Spara</td>
<td>6.05</td>
<td>5,424.45</td>
<td>7.01</td>
<td>2,248.71</td>
<td>12,148.68</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Scere</td>
<td>6.05</td>
<td>5,424.45</td>
<td>7.01</td>
<td>2,212.71</td>
<td>13,716.93</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Athal</td>
<td>6.05</td>
<td>5,424.45</td>
<td>7.01</td>
<td>2,868.71</td>
<td>15,496.11</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Sson</td>
<td>6.05</td>
<td>10,800.45</td>
<td>7.01</td>
<td>2,392.71</td>
<td>18,875.41</td>
<td>2,074.62</td>
<td>14.95</td>
</tr>
<tr>
<td>Bact</td>
<td>6.05</td>
<td>10,800.45</td>
<td>7.01</td>
<td>5,407.71 ∗</td>
<td>20,571.50 ∗</td>
<td>2,071.92</td>
<td>16.11</td>
</tr>
<tr>
<td>Hsap</td>
<td>6.23</td>
<td>10,802.20</td>
<td>7.16</td>
<td>3,254.96 ∗</td>
<td>26,121.55 ∗</td>
<td>2,071.92</td>
<td>16.11</td>
</tr>
</tbody>
</table>

(b) dataset-rep

Table 4.7: Approximate maximum memory used during compression in Mbytes by each of the general-purpose compressors. The ∗ symbols indicate that these sequences were compressed in blocks. Refer to Table 4.3 for details.

gzip, bzip2 and ppmd were unsuitable for compressing large repetitive collections. In our discussion in Chapter 3, we expected that Re-pair would produce the best compressed sizes, followed by Sequitur and 7-Zip. Although Re-pair outperformed Sequitur for highly repetitive collections, surprisingly, 7-Zip performed significantly better than the other two tools. The 1 Gbyte sliding window effectively implements global repeat detection for most collections in dataset-rep and the LZMA encoder is able to compactly represent the compressed output. Even the significantly larger 12 Gbase Hsap collection was compressed effectively by 7-Zip, since the copies of the same chromosome were placed adjacently. Therefore most repeats were within the sliding window allowing 7-Zip to detect these repeats.

Since 7-Zip is able to detect repeats across multiple files, we also conducted a trivial experiment to illustrate the impact on the 7-Zip compression performance when the sequence file ordering is changed. We used the Spara collection of dataset-rep and set the sliding window to 12 Mbytes, the size of an individual S. paradoxus genome. We used two file orderings; one is a randomised ordering, and the other is based on the sequence similarity ordering determined by our RLZ algorithm as shown in Figure 8.1. The compressed sizes were 12.26, and 5.57 Mbytes,
respectively. This shows that the compression performance of 7-Zip varies significantly based on the file ordering. If the ordering based on sequence similarity is known in advance, then excellent results can be achieved. However, the results are less promising if no such information is available in advance. Although this is a contrived example and for collections smaller than a 1 Gbase, 7-Zip could detect repeats globally, the file ordering knowledge is essential for larger collections to achieve the best possible compression. Our COMRAD algorithm provides a solution to this issue in Chapter 5.

In this section we evaluated the compression performance of some well-known general-purpose compression algorithm implementations. Contrary to what many of the authors of DNA compression tools have stated [Grumbach and Tahi, 1993], certain general-purpose compressors like 7-Zip produced good results for compressing DNA sequences. In the next section, we evaluate the compression performance of some DNA-specific compression tools for the same test datasets. Then in Section 4.5, we compare the results of the general-purpose and DNA compression tools.

4.4 DNA compression algorithm evaluation

In this section we evaluate the compression performance of some publicly available implementations of DNA compression algorithms. Unfortunately most tools described in Section 3.3 are either unavailable or cannot be used on current platforms as a result of not being actively maintained. Below we describe each tool used for our experiments, and the parameters used, followed by the results. The sources of the compression algorithm implementations are presented in Appendix A.

4.4.1 Algorithm parameters

Sections 3.3.1 and 3.3.2 described DNA compression algorithms that detect exact and approximate repeats, respectively, and in this section, we aim to evaluate the implementations of these algorithms for compressing the sequences and sequence collections in our test datasets. However, from the implementations that detect exact repeats, we were only able to use dna-x, and only GenCompress, DNACompress, and XMCompress from the implementations that detect approximate repeats. The parameters used for each tool during the experiments are as follows:

dna-x: As explained in Section 3.3.1, the dna-x implementation has four versions, each of which uses different encoding strategies. The best results are produced by dna-3, and it was invoked with the command dnaX -c7 -b20 -s0 -o3, where the options were specified to use Huffman and canonical Huffman encoding for the lengths of the matched and unmatched prefixes, and to use an order-3 arithmetic encoder for the unmatched prefix.
GenCompress: GenCompress allows the user to specify the maximum number of edit operations $b$ permitted on a string of length $k$, with default values $k=12$ and $b=3$. Ideally, $k$ and $b$ should be chosen to reflect the approximate repeat properties of sequences being compressed. We used the default values, since we do not know the approximate repeat properties of each dataset.

XMCompress: XMCompress has many parameters that can be set to tune the experts. Since the default parameters are tuned for large sequences (10 Mbases or more\(^1\)), we use the default parameters to compress the sequences in our test datasets.

DNACompress: The DNACompress implementation is publicly available only as an executable to run under the Windows operating system. The experiments were run on a machine with a 2.6GHz Intel Core 2 Duo processor and 4 Gbytes of RAM running Windows XP. As a result we only compare the compressed size results of DNACompress with other tools, but do not compare the compression and decompression speeds or memory usage. The DNACompress implementation also has many parameters that can be specified. Even though we attempted to use the parameters stated by the authors in their experiments [Chen et al., 2002], the software produced the same results without responding to any parameter changes. This is most likely due to a software error, or the demo version of the PatternHunter alignment tool required by DNACompress does not allow the parameters to be changed. Therefore, we used the following default parameters as displayed by the software: $-r=1$, $-q=-4$, $-G=4$, $-E=3$, $-h=3$, $-N=1024$, $-M=24$.

For the Cfact, DNASequitur, DNAPack, CTW+LZ, NML and GeNML algorithms, there were no publicly available implementations. The Biocompress and Biocompress-2 implementations were unable to run on a 64-bit machine. On a 32-bit machine, the output produced was a series of 0 and 1 symbols in plain text, and it was unclear what this output was. We used CDNA to compress a set of small sequences that are a few kilobases in size. The tool estimated good compression results, although it does not physically compress sequences. However, due to the slow speed (over half an hour to compress a small sequence), it will not scale for large sequence compression, so we do not attempt to experiment with this tool for large sequences.

In the next section, we present the compression results for the tools discussed above, and in Section 4.5, we compare the performance of the general-purpose compression tools with the DNA compression tools.

\(^1\)Personal correspondence with the first author.
4.4.2 Results

Below we evaluate the compression performance of the DNA compression tools listed above in terms of the three criteria, compressed size, compression and decompression speed, and memory usage.

Compressed size

Table 4.8 reports the compressed sizes produced by each tool with the second column containing the size produced by a simple 2 bpb encoding. Although GenCompress produced good results, it did not scale for compressing large sequences. The 12 Mbase ScereRef sequence was compressed in 21 minutes, but the 23 Mbase PfalcRef sequence was not compressed even after 9 hours. Similarly, the Hemo collection was compressed in 3 hours, but the Mito collection was not compressed even after 66 hours. This shows that the time complexity is not linear in the length of the sequence, so we did not attempt to compress larger sequences with GenCompress. For Hemo, GenCompress performed almost as well as XMCompress. DNACompress could not compress the two largest sequences in dataset-sin and the six largest collections in dataset-rep, and for the remaining sequences, the compression performance was not as good as expected, most likely due to the software error mentioned earlier.

XMCompress and dna-x were the only tools that were able to compress most of the sequences and collections. As expected, the exact repeat detecting tool, dna-x, produced larger sizes than the approximate repeat detecting tool, XMCompress. The results of dna-x were still promising. Both tools significantly out-performed the simple 2 bpb representation. Neither dna-x nor XMCompress can scale to compress large collections; dna-x in terms of the run time, and XMCompress in terms of the memory use and runtime. Therefore, a few of the largest sequences and collections had to be compressed in blocks. We did not compress the HsapRef sequence with dna-x, since even after splitting the sequence into 6 equal-sized blocks, a single block required over 11 hours to compress. This indicates that the tool cannot scale for compressing large sequences or collections. We also did not attempt to compress the Hsap collection with these tools.

The results show that both tools perform well for moderately large collections. The average number of bits used by each tool to encode a nucleotide, as shown in Table 4.9, also confirms these results. All tools used less bits than predicted by the 0-order entropy. XMCompress clearly produced the best results and its compressed size for the HsapRef genome is the best result that we are aware of in the literature.

Compression and decompression speed

Next we analyse the time taken by each tool to compress the test sequences and collections, and the compression times are reported in Table 4.10. The fastest compression...
### 4.4. DNA COMPRESSION ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>Dataset</th>
<th>2bpb</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>DNACompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>2.00</td>
<td>1.06</td>
<td>1.06</td>
<td>1.09</td>
<td>1.06</td>
</tr>
<tr>
<td>ScereRef</td>
<td>3.00</td>
<td>2.68</td>
<td>2.67</td>
<td>2.82</td>
<td>2.65</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>6.00</td>
<td>4.44</td>
<td>—</td>
<td>4.60</td>
<td>4.18</td>
</tr>
<tr>
<td>CelegRef</td>
<td>24.00</td>
<td>21.33</td>
<td>—</td>
<td>22.63</td>
<td>20.47</td>
</tr>
<tr>
<td>AthalRef</td>
<td>29.00</td>
<td>25.58</td>
<td>—</td>
<td>27.27</td>
<td>23.97</td>
</tr>
<tr>
<td>DmelRef</td>
<td>41.00</td>
<td>29.96</td>
<td>—</td>
<td>38.25</td>
<td>28.40</td>
</tr>
<tr>
<td>GgalRef</td>
<td>247.00</td>
<td>218.23*</td>
<td>—</td>
<td>214.21*</td>
<td></td>
</tr>
<tr>
<td>HsapRef</td>
<td>739.00</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>554.94*</td>
</tr>
</tbody>
</table>

**Table 4.8:** Compressed sizes in Mbytes produced by the DNA compression tools. The second column contains the size of a 2 bpb encoding, and remaining columns contain results for each tool. The * symbol indicates that a sequence was compressed in blocks. For dataset-sin, GgalRef was divided into two equal-sized blocks for dna-x and XMCompress. HsapRef was divided into 6 equal-sized blocks for XMCompress. For dataset-rep, dna-x compressed Sson and Bact collections in two and four equal-sized blocks, respectively. XMCompress compressed the Bact collection in four-equal sized blocks. The — symbol indicates that a tool could not compress a sequence.

<table>
<thead>
<tr>
<th>Coll.</th>
<th>2bpb</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>DNACompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>2.00</td>
<td>0.69</td>
<td>0.60</td>
<td>1.68</td>
<td>0.52</td>
</tr>
<tr>
<td>Mito</td>
<td>7.00</td>
<td>4.68</td>
<td>—</td>
<td>5.78</td>
<td>3.04</td>
</tr>
<tr>
<td>InfI</td>
<td>27.00</td>
<td>3.67</td>
<td>—</td>
<td>25.83</td>
<td>1.84</td>
</tr>
<tr>
<td>Ecol</td>
<td>40.00</td>
<td>5.67</td>
<td>—</td>
<td>38.57</td>
<td>3.69</td>
</tr>
<tr>
<td>Spara</td>
<td>103.00</td>
<td>7.79</td>
<td>—</td>
<td>—</td>
<td>5.34</td>
</tr>
<tr>
<td>Scere</td>
<td>116.00</td>
<td>6.28</td>
<td>—</td>
<td>—</td>
<td>4.27</td>
</tr>
<tr>
<td>Athal</td>
<td>121.00</td>
<td>33.61</td>
<td>—</td>
<td>—</td>
<td>30.68</td>
</tr>
<tr>
<td>Sson</td>
<td>231.00</td>
<td>8.17*</td>
<td>—</td>
<td>—</td>
<td>6.54</td>
</tr>
<tr>
<td>Bact</td>
<td>661.00</td>
<td>518.98*</td>
<td>—</td>
<td>—</td>
<td>502.02*</td>
</tr>
</tbody>
</table>

**Table 4.8 (continued):** Compressed sizes in Mbytes produced by the DNA compression tools. The second column contains the size of a 2 bpb encoding, and remaining columns contain results for each tool. The * symbol indicates that a sequence was compressed in blocks. For dataset-sin, GgalRef was divided into two equal-sized blocks for dna-x and XMCompress. HsapRef was divided into 6 equal-sized blocks for XMCompress. For dataset-rep, dna-x compressed Sson and Bact collections in two and four equal-sized blocks, respectively. XMCompress compressed the Bact collection in four-equal sized blocks. The — symbol indicates that a tool could not compress a sequence.

The decompression speeds for the DNA compression tools are also reported in Table 4.10. GenCompress was able to decompress much faster than it could compress. On the other hand, XMCompress is slower to decompress than it is to compress, but this is typical for statistical compressors. XMCompress could not decompress the two...
Table 4.9: Average number of bits used by each DNA compression tool to encode a nucleotide. The second column contains the 0-order entropy of each collection, and the remaining columns contain the results for each tool. The — symbol indicates that a tool was unable to compress the collection or it could not compress in a feasible amount of time. The * symbol indicates that a collection was compressed in blocks. Refer to Table 4.8 for more information.

largest sequences and the six largest collections due to a software issue. Similarly, dna-x could not decompress the Spara and Scere collections. Overall, dna-x has the fastest decompressor out of the three tools.

**Memory usage**

Table 4.11 reports the approximate maximum memory used by the DNA compression tools during compression. The most memory-efficient tool was dna-x, which only used a small amount of memory overhead compared to the size of the input sequence. For the sequences it could compress, GenCompress used memory that is several times larger than the input size; for example, 472 Mbytes were required to compress the 7 Mbase Hemo collection. XMCompress also required a significant amount of memory to compress. Even to compress the 12 Mbase ScereRef sequence, nearly 3 Gbytes of memory was used. Recall that the large memory requirements of XMCompress is the reason that GgalRef and HsapRef sequences were compressed in blocks. Therefore, XMCompress does not scale for compressing large sequences, in
### 4.4. DNA COMPRESSION ALGORITHM EVALUATION

<table>
<thead>
<tr>
<th>Dataset</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>4</td>
<td>2</td>
<td>250</td>
</tr>
<tr>
<td>ScereRef</td>
<td>11</td>
<td>6</td>
<td>1,300</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>36</td>
<td>10</td>
<td>—</td>
</tr>
<tr>
<td>ClegRef</td>
<td>93</td>
<td>50</td>
<td>—</td>
</tr>
<tr>
<td>AthalRef</td>
<td>114</td>
<td>60</td>
<td>—</td>
</tr>
<tr>
<td>DmelRef</td>
<td>2,301</td>
<td>68</td>
<td>—</td>
</tr>
<tr>
<td>GgalRef</td>
<td>52,966*</td>
<td>527*</td>
<td>—</td>
</tr>
<tr>
<td>HsapRef</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

(a) dataset-sin

<table>
<thead>
<tr>
<th>Collection</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>3</td>
<td>1</td>
<td>10,660</td>
</tr>
<tr>
<td>Mito</td>
<td>17</td>
<td>9</td>
<td>—</td>
</tr>
<tr>
<td>Inf1</td>
<td>45</td>
<td>4</td>
<td>—</td>
</tr>
<tr>
<td>Ecol</td>
<td>31</td>
<td>12</td>
<td>—</td>
</tr>
<tr>
<td>Spara</td>
<td>12,335</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Scere</td>
<td>53,168</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Athal</td>
<td>25,638</td>
<td>89</td>
<td>—</td>
</tr>
<tr>
<td>Sson</td>
<td>4,925*</td>
<td>34*</td>
<td>—</td>
</tr>
<tr>
<td>Bact</td>
<td>2,908*</td>
<td>1,263*</td>
<td>—</td>
</tr>
</tbody>
</table>

(b) dataset-rep

Table 4.10: Compression times in seconds required to compress the sequences by each DNA compression tool. The — symbol indicates that a tool was unable to compress or decompressed the sequence or took too long to be feasible. The * symbol indicates that a sequence was compressed or decompressed in blocks. Refer to Table 4.8 for more information.

In terms of the compression and decompression speed, and memory use.

In summary, dna-x and XMCompress were the only tools we experimented with that were able to compress most of our test sequences. In terms of the compression speed, GenCompress does not scale to even compress genomes that are only a few megabases long. DNACompress was able to compress slightly larger sequences than GenCompress. But it was still unable to compress a sequence larger than 500 Mbases. Sequences that are less than a gigabase in length can be compressed effectively with dna-x. For larger sequences, the tool does not scale in terms of the compression time. The best compression results were produced by XMCompress. Unfortunately the tool used significant amounts of memory for sequences that are a gigabase or longer. Although the compression speed of XMCompress is faster than that of dna-x, it is still slow and is impractical for compressing a large collection. If the compression speed is not important, then XMCompress is the best DNA compression tool available.
### Table 4.11: Approximate maximum memory used in Mbytes by the DNA compression tools during compression.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>14.52</td>
<td>55.03</td>
<td>757.33</td>
</tr>
<tr>
<td>ScereRef</td>
<td>24.57</td>
<td>156.92</td>
<td>2,729.68</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>39.39</td>
<td>—</td>
<td>4,388.31</td>
</tr>
<tr>
<td>C elegRef</td>
<td>142.21</td>
<td>—</td>
<td>6,920.28</td>
</tr>
<tr>
<td>AthalRef</td>
<td>167.41</td>
<td>—</td>
<td>4,684.81</td>
</tr>
<tr>
<td>DmelRef</td>
<td>233.62</td>
<td>—</td>
<td>5,220.89</td>
</tr>
<tr>
<td>GgalRef</td>
<td>697.18</td>
<td>—       *</td>
<td>9,609.68*</td>
</tr>
<tr>
<td>HsapRef</td>
<td>—</td>
<td>—</td>
<td>7,532.37*</td>
</tr>
</tbody>
</table>

(a) dataset-sin

<table>
<thead>
<tr>
<th>Coll.</th>
<th>dna-x</th>
<th>GenCompress</th>
<th>XMCompress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>18.18</td>
<td>450.71</td>
<td>965.59</td>
</tr>
<tr>
<td>Mito</td>
<td>42.05</td>
<td>—</td>
<td>2,497.16</td>
</tr>
<tr>
<td>Infl</td>
<td>158.72</td>
<td>—</td>
<td>2,673.64</td>
</tr>
<tr>
<td>Ecol</td>
<td>228.49</td>
<td>—</td>
<td>5,253.47</td>
</tr>
<tr>
<td>Spara</td>
<td>581.46</td>
<td>—</td>
<td>7,059.82</td>
</tr>
<tr>
<td>Scere</td>
<td>657.04</td>
<td>—</td>
<td>7,377.91</td>
</tr>
<tr>
<td>Athal</td>
<td>684.80</td>
<td>—</td>
<td>7,387.86</td>
</tr>
<tr>
<td>Sson</td>
<td>653.51</td>
<td>—       *</td>
<td>11,091.04</td>
</tr>
<tr>
<td>Bact</td>
<td>933.10</td>
<td>—       *</td>
<td>10,153.27</td>
</tr>
</tbody>
</table>

(b) dataset-ref

The * symbol indicates that a sequence was compressed in blocks. Refer to Table 4.8 for more information.

### 4.5 Discussion

We now compare the compression results produced by general-purpose compression algorithms with those of the DNA compression algorithms, to determine whether DNA-specific compression is necessary. Recall that since Grumbach and Tahi [1993], the authors of DNA compression algorithms have argued that general-purpose compression algorithms are unable to compress DNA sequences better than a simple 2 bpb representation of the sequences [Chen et al., 2000; Manzini and Rastero, 2004]. We used our test datasets to evaluate this hypothesis.

The results for dataset-sin confirm the advantages of DNA-specific compression algorithms. Recall that dataset-sin contained a set of complete genome sequences and the repeat analysis in Section 4.1.3 confirmed that the amount of exact repeats within these genomes is small. The results in Section 4.3.2 showed that general-purpose compression tools like gzip, compress, bzip2 and Re-pair were unable to compress the genomes in dataset-sin to use less than 2 bpb. However, 7-Zip and Sequitur were able to produce better results than the 2 bpb baseline. In fact, 7-Zip was excellent at compressing these relatively non-repetitive genomes, given that it
4.5. DISCUSSION

does not detect approximate repeats. Unsurprisingly, as shown in Section 4.4.2, for the genomes in dataset-sin, the best performing DNA compression tools, dna-x and XMCompress out-performed the best performing general-purpose compression tools, 7-Zip and Sequitur. The XMCompress results were far better than those produced by 7-Zip. This shows that single sequences contain approximate repeats and compression algorithms need to detect these repeats to effectively compress single sequences. One of the observations made in this experiment was that DNA compression algorithms are not implemented to scale, as most of the tools were unable to compress the larger genomes, either due to long execution times spanning many days, or due to high memory use. Therefore, scalability is an important issue to consider when we implement our own DNA compression algorithms.

The results for dataset-sin confirmed that DNA-specific compression algorithms are indeed better at single-sequence compression than even the best general-purpose compression algorithms. The results for dataset-rep in Section 4.3.2 showed that some general-purpose compression tools like gzip, compress and bzip2 produced larger compressed sizes than using a simple 2 bpb encoding in the presence of high levels of repetition. However, tools such as 7-Zip, Sequitur and Re-pair were able to compress the repetitive collections well. Given the high level of exact repeats in these collections, as confirmed by our repeat analysis in Section 4.1.3, it is unsurprising that algorithms detecting global repeats are able to compress these collections well. When comparing these results to those of DNA compression tools in Section 4.4.2, it is evident that 7-Zip performs nearly as well as XMCompress. XMCompress tends to achieve slightly better results since it can detect the small amount of approximate repeats in these sequences.

The results for dataset-rep showed that general-purpose compression algorithms are just as effective as DNA compression algorithms for compressing highly repetitive sequence collections. However, DNA compression algorithms handle DNA-specific features to achieve better compression and this is confirmed by our experiments. Therefore, it appears to be a correct assumption that DNA-specific compression algorithms are valuable. For large repetitive sequence collections, it is sufficient to detect exact repeats, since the gain in compression by using approximate repeat detection is small compared to the increase in compression time and memory use. Therefore, the DNA compression algorithms we introduce in this thesis use exact repeat detection. Unlike the existing DNA compression algorithms, our algorithms will be specifically designed to handle large collections.

Most compression tools we experimented with in this chapter cannot detect repeats across sequences. Therefore, all individual sequences in a collection were concatenated to form a single sequence, such that the tools could detect repeats that occur across sequences. Specifically, for the four human genomes, we ensured that the four copies of the same chromosome were adjacent to each other, and as a result, most tools were able to compress the collection exceptionally well. However,
it may not necessarily be possible to pre-process collections in a manner that allows single-sequence compression tools to detect repeats. The results would have been quite different had we compressed each collection a single sequence or genome at a time. It would be especially difficult to pre-process the data in this manner if we were to compress hundreds or thousands of human genomes. Furthermore, access to individual genomes in the compressed collection incurs a large overhead as the entire collection needs to be decompressed to access a single sequence. If the sequences were compressed individually, this would not be an issue. Therefore, we introduce a DNA compression algorithm in the next chapter that is able to detect repeats across multiple sequences without requiring that the sequences be pre-processed into a single file. The algorithm will detect repeats across multiple sequences regardless of the order in which the sequences are analysed. We believe this is an important feature for compressing large sequence collections.

4.6 Chapter summary

In this chapter, we analysed the performance of general-purpose and DNA-specific compression algorithm implementations for compressing DNA sequences and highly repetitive sequence collections. Overall, we found that DNA compression tools like XMCompress and dna-x, were more effective at compressing DNA than most general-purpose compression tools. The exception for general-purpose tools was 7-Zip, which was able to produce excellent compression results for single genomes and repetitive collections. The tools Re-pair and Sequitur, which detect repeats globally were also very effective at compressing repetitive sequence collections.

A key observation made in the chapter was the lack of scalability of some of the existing tools for compressing large sequences. XMCompress and dna-x did not scale in terms of the compression speed for large collections, while XMCompress, Sequitur and Re-pair did not scale in terms of memory use. As a result, large collections must be compressed in blocks, which hinders the compression performance given that repeats cannot be detected globally.

In the next two chapters, we introduce our own compression algorithms that address some of the scalability issues. In Chapter 5, we introduce a disk-based semi-static DNA compression algorithm known as COMRAD that detects repeats globally. Although adaptive algorithms like 7-Zip, Sequitur, dna-x and XMCompress compress well, we are interested in determining whether semi-static algorithms make better substitution decisions than the adaptive algorithms. COMRAD is disk-based which allows it to scale for compressing large collections. Then in Chapter 6, we introduce our RLZ algorithm that compresses sequences with respect to a reference. Unlike COMRAD which makes no assumptions about the types of collections being compressed, RLZ assumes that the sequences being compressed are very similar to the chosen reference, making it ideal for compressing repetitive sequence collections.
Chapter 5

Efficient genome storage with COMRAD*

The analysis in Chapter 4 showed that although 7-Zip, an adaptive algorithm, produced the best compression results, Re-pair, a semi-static algorithm, produced the next best results for highly-repetitive collections. The good performance of Re-pair can be attributed to the fact that repeats are selected carefully based on frequencies. Global repeat detection also allows the detection of repeats which are typically far apart in DNA collections. For larger collections, we can expect Re-pair to maintain its good performance, unlike 7-Zip, which is not likely to maintain the same performance, given the limitations on the sliding window size. Even for the Hsap collection, 7-Zip would not have produced such good results, had all the sequences of the same chromosome not been placed alongside each other before compression. The results produced by Re-pair would not have been affected by such a change. DNA compression algorithms like XMCompress that detect approximate repeats achieved better compression than Re-pair. However, the resource usage was high, both in terms of memory use and the time taken to compress and decompress. This was particularly an issue when compressing large collections, where many days were required to produce a result. Re-pair also used a significant amount of memory during compression, but was much faster at compressing large collections and even faster during decompression compared to XMCompress.

Based on the observations from Chapter 4, we hypothesise that a DNA-specific semi-static dictionary compression algorithm will be ideal for compressing large repetitive DNA collections. Such an algorithm could achieve compression results that are comparable to DNA compression algorithms, and with careful implementation considerations, use less memory and have fast compression and decompression speeds for large input DNA collections.

In this chapter, we propose a semi-static dictionary compression algorithm for DNA sequences known as COMRAD (COMpression using RedundAncy of Dna). The

*An earlier version of research presented in this chapter was published in Kuruppu et al. [2012].
algorithm is suitable for compressing DNA sequences containing repetitions. However, given that this thesis aims to explore efficient storage mechanisms for the output of genome sequencing projects, the COMRAD algorithm is tailored for compressing large repetitive multi-sequence DNA collections. With multi-sequence compression, repeats that occur across the sequences in a collection can be considered to make the best substitution decisions. The main focus of the COMRAD algorithm is on the modelling technique used to construct the dictionary of global repeats, rather than on the encoding step of the algorithm. The ability to efficiently detect long repeats that occur across multiple sequences is more significant to the final compression result. Once a good model is available, the encoding techniques can then be optimised to achieve the final compression result. Another important aspect of COMRAD is that it is a disk-based algorithm, where the entire collection does not need to be stored in memory. In this chapter, we present the COMRAD algorithm, a discussion of its properties and a practical analysis of its performance.

The COMRAD algorithm is based on the RAY algorithm presented in Section 3.2.2. In Section 5.1 we explain the reasons for basing COMRAD on RAY, while also presenting a detailed description of the improvements made to RAY and the reasons behind the improvements. In Section 5.2, we describe the algorithm in detail, followed by experimental results which compare the performance of COMRAD to the existing general-purpose and DNA compression algorithms from Chapter 4. We also discuss the various parameters of COMRAD that can be adjusted.

An advantage that dictionary compression algorithms have over the statistical compressors like XMCompress is the ability to invoke random access queries on the compressed data without decompressing the entire collection. This is a useful feature for large collections, since performing queries on the compressed collection can be more efficient than performing the same queries on the original collection. In Section 5.3, we present the algorithm for implementing the random access feature on COMRAD-compressed sequences, followed by a comparison of the query performance of COMRAD with existing self-indexing algorithms that implement this query.

5.1 Modifications to RAY

Recall from Section 3.2.2 that RAY is a semi-static algorithm that compresses the input by replacing frequent symbol pairs with non-terminals over multiple passes through the input. RAY is similar to Re-pair, and the main difference is that during an iteration of RAY, multiple distinct symbol pairs can be substituted by non-terminals, which reduces the number of iterations of the algorithm. Therefore, we base our algorithm on RAY rather than Re-pair. The improvements are as follows:
5.1. MODIFICATIONS TO RAY

1. Restrict the input alphabet to the extended DNA alphabet

We restrict the valid input alphabet to the symbols from the extended DNA alphabet $\Sigma$. While theoretically the COMRAD algorithm can be applied to any collection, in practice, the small size of the alphabet can be exploited in the implementation to make the algorithm more memory-efficient.

2. Substitute frequent $L$-mers in the first iteration

Unlike RAY which substitutes symbol pairs, COMRAD allows substrings of length $L$ to be substituted in the first iteration. As we discussed in Section 2.1.4, the repeated substrings in DNA are much longer than the repeats observed in standard text documents. Therefore, replacing only pairs of symbols at a time would require a large number of iterations to detect long repeats. By substituting larger substrings in the first iteration, we can reduce the number of iterations of the algorithm, potentially improving the compression speed. Typically we set $L$ to 15 or 16 to detect genuine repeats rather than substrings that are repeated by chance. The drawback of substituting longer substrings is that their frequency counts need to be stored, which is more resource-intensive given the larger number of distinct $L$-mers. We use some optimisations based on the small alphabet size in our implementation to counter this problem. Another issue is that repeats of lengths smaller than $L$ are not detected by this method, although the $L$ parameter can be adjusted to detect shorter repeats.

3. Substitute substrings matching patterns in subsequent iterations

After the first iteration, the input sequence consists of terminals and non-terminals. We can continue to substitute $L$-mers, now consist of terminals and non-terminals. However, this is not feasible, since the first iteration introduces many non-terminals, resulting in at most $(z + |\Sigma|)^L$ distinct $L$-mers, where $z$ is the number of non-terminals introduced in the first iteration and $|\Sigma|$ is the size of alphabet $\Sigma$. We can adopt the RAY approach of replacing symbol pairs, however this requires more iterations for repeats that span thousands to millions of bases. For a disk-based algorithm like COMRAD, this cost can be significant when compressing large collections. Therefore, we define a set of patterns that capture substrings of length two or more that satisfy certain terminal and non-terminal combinations, such that the number of iterations are reduced without increasing the memory use significantly. We propose that repeated substrings matching the following patterns should be substituted by COMRAD: $AXB$, $AX$ and $XB$, where $A$ and $B$ are non-terminals created in earlier iterations, and $X$ is a substring of nucleotides ranging in length from 0 to some maximum length $m$. The pattern $AXB$ substitutes repeats, where the two ends of the repeat were substituted by non-terminals in an earlier iteration, but the substring in between was too short to be detected. The patterns $AX$ and $XB$ substitute shorter repeats that are adjacent to previously substituted longer repeats.
4. Reverse complement detection

As discussed earlier in Section 2.1.4, repeats in DNA can be standard repeats as well as reverse complement repeats. While these are not very common in most sequences, it is still an important type of repeat to detect. The inclusion of reverse complement repeat detection was first suggested by Grumbach and Tahi [1993] and nearly all DNA compression algorithms since then detect reverse complement repeats. Therefore, we also include reverse complement repeat detection in COMRAD.

5. Selecting repeated substrings to be replaced

The next modification is made to the candidate selection algorithm of RAY. The RAY substitution algorithm is a two step process, where in the first step, triplets of symbols are checked to determine which symbol pair is more suitable for replacement. Then the chosen symbol pairs are substituted with non-terminals. While this algorithm considers the relative frequencies of overlapping symbol pairs, it does not consider the relative frequencies of each distinct symbol pair with respect to the other symbol pairs. To consider the relative frequencies, we sort the symbol pairs in the sequence from highest to lowest frequency. Then the non-overlapping symbol pairs from the most frequent to the least frequent are substituted. With this method, repeats that occur with a high frequency are favoured for replacement.

6. Multi-sequence compression

Most DNA and general-purpose compression algorithms that we analysed in Chapter 4 compress a single input at a time. However, for compressing multiple related sequences, repeats that exist across the sequences need to be considered to make the best substitution decisions. As we did in Chapter 4 for the collections in DATASET-REP, we can concatenate the sequences, but we may need to store the entire concatenated sequence in memory, which may not be possible for large collections. Instead of concatenating the sequences, we keep the sequences on disk and only process one sequence at a time. At each iteration, each sequence is read in to memory to store the global substring frequency counts. Then each sequence is read again to perform the substitutions, and the modified sequence is written back to disk. In this manner, the COMRAD algorithm keeps track of the global frequency counts at each iteration and only holds one sequence in memory at a time. Therefore, even if large sequences are divided into smaller blocks, the repeats across the blocks are still detected and compressed. The disconnect between the repeat detection and substitution steps also means that the compression results are not affected by the order in which the sequences are compressed. Note that even though the substitution decisions in a sequence are made independent to the decisions of other sequences, since the same frequency dictionary is used for all sequences, it is likely that the same substitution decisions will be made in the identical repeat regions that occur across sequences.
5.2 COMRAD compression

The COMRAD algorithm makes few assumptions about the DNA collections that it compresses. The collections can contain nucleotides from the extended DNA alphabet. No assumptions are made about the repeat properties of the sequences, and collections can contain multiple sequences from the same or different species. However, COMRAD is most suitable for compressing highly repetitive DNA collections, such as genomes from individuals or strains from the same species, where the repeats are long and mostly exact. For non-repetitive collections, COMRAD may increase the compressed size to above the value predicted by the 0-order entropy of the collection.

Below we describe the COMRAD algorithm in detail. We begin with a specification of the input parameters. Then the detailed compression and decompression algorithms are presented, followed by the encoding techniques used to compress the output of the algorithm for storage.

5.2.1 Algorithm parameters

The COMRAD algorithm has three parameters; the starting length \( L \) of repeats, the minimum frequency threshold \( F \) that substrings should satisfy if they are to be substituted by a non-terminal, and the set of patterns that can be detected from the second iteration onwards. Below we explain each parameter in detail.

**Initial length:** The initial substring length \( L \), is the substring length to be used in the first iteration. The justification for including this parameter was discussed in Section 5.1. Given a value \( L \), the algorithm counts the frequencies of all distinct \( L \)-mers in the input sequences and substitutes the most frequent \( L \)-mers with non-terminals. Therefore, all repeats that are substituted in the first iteration are of length \( L \). Furthermore, the COMRAD algorithm will only substitute repeated substrings of length \( \geq L \). Intuitively, the higher the value of \( L \), the lower the substitutions made, since the number of repeats with at least a length of \( L \) will be lower. On the other hand, the lower the value of \( L \), the more substitutions that will be made, but a greater number of iterations are required to complete the compression.

The default value is \( L = 16 \), since the aim of the first iteration is to identify the legitimate repeats in the collection and avoid recognising \( L \)-mers that occurred by chance as repeats. With \( L = 16 \), there is a 1 in \( 4^{16} \) probability of an \( L \)-mer occurring by chance. The higher the \( L \) parameter, the less likely that an \( L \)-mer will occur by chance, and more likely that a repeated \( L \)-mer is a legitimate repeat. The implementation details considering this parameter choice are discussed in Appendix B. We also perform experiments in Section 5.2.6 to determine the effect on compression when the parameter \( L \) is changed.
Substrings frequency: The minimum frequency threshold that a substring must satisfy before it is eligible to be substituted is defined as $F$. At each iteration of the algorithm, the frequencies of all distinct substrings are found, and only those with a frequency of at least $F$ are considered for substitution. The algorithm terminates when no substrings remain with a frequency of at least $F$. The higher the value of $F$, the higher the compressed size, and lower the time taken to compress, since most repeats will not satisfy the frequency threshold. We set the default value of $F = 4$ as this is the value found experimentally by the authors of RAY to be the value that achieves a net space saving. In Section 5.2.6, we illustrate the effects on compression when the parameter $F$ is changed.

Patterns to detect: In Section 5.1, the third modification stated that from the second iteration onwards, substrings matching patterns of the form $AXB, AX'$ and $X'B$ will be detected and substituted, where $A$ and $B$ are non-terminals created in earlier iterations, and $X$ and $X'$ are substrings of nucleotides ranging in length from 0 to some maximum length $m$ and $m'$, respectively. Currently, the COMRAD algorithm defines a set of patterns to detect such that a given substring will match to only one of the patterns or none. The current implementation does not allow the user to specify the patterns. The defined patterns are as follows:

1. $AB$
2. $AxB$
3. $AxxB$
4. $AxxxB$
5. $AxxxBx$
6. $AxxxBxx$
7. $Ax$xxx

$A$ and $B$ are non-terminals introduced in a previous iteration and $x_i$ is a nucleotide from alphabet $\Sigma$. The first four patterns are designed to detect pairs of repeated non-terminals containing 0–4 nucleotides in-between. The last two patterns capture the ends of repeats too short to be captured by an $L$-mer and were not in the vicinity of other non-terminals. By detecting certain repeated substrings of lengths 2–6, the overall number of iterations of the algorithm are reduced.

Figure 5.1 illustrates some examples of detecting and replacing repeated substrings using the above patterns. The initial substring length $L = 8$ and frequency threshold $F = 2$ were chosen for this example. The first example illustrates how pattern 1 is used. The sequence contains two repeats of the substring acgattgcacgattgc. In the first iteration, each acgattgc substring is replaced by a rule resulting in the sequence aAAttAAg. Then pattern 1 captures the repeat AA to achieve the final encoding of aBttBg.

The second example illustrates how pattern 3 is used. The sequence contains two repeats of the substring acgattgcgacgattgc. Because the sequence contains 5 occurrences of the substring acgattgc, these are replaced in the first iteration resulting in the sequence aAggAAttAaggAgt. Then pattern 3 captures the repeat AggA and this is replaced to produce the final sequence aBttBgAt.
The third example illustrates how pattern 6 is used. The sequence contains two repeats of the substring \texttt{acgattgctccta}. The sequence contains 3 occurrences of the substring \texttt{acgattgc}, these are replaced in the first iteration to produce the sequence \texttt{AtcctagggAtcctagctAt}. Then pattern 6 captures the repeat \texttt{Atccta} and this is replaced to produce the final sequence \texttt{BgggBgctAt}. Note that after the first replacement, the substring \texttt{Atcctag} is repeated twice. However, pattern 6 cannot detect the last \texttt{g} nucleotide of the repeat because it is restricted to only detect at most 5 nucleotides that trail a non-terminal. We agree this is an arbitrary limitation and it is a drawback of the algorithm that repeats of a certain length are not completely substituted. However, our aim is to detect as many of the long repeats as possible, rather than detect all repeats. While there may be slight gains or losses in the compressed size by adjusting the number of nucleotides detected in a pattern, provided most of the nucleotides in long repeats are replaced, the difference in the compressed size will not be significant.

The final example illustrates how pattern 7 is used. Once again, even though the substring after the first iteration contains two occurrences of the substring \texttt{tcctatA}, pattern 7 can only detect the substring \texttt{cctatA}.

Evidently, many other patterns could be introduced to reduce the number of iterations and to detect more repeats, but at the cost of increased time to compress due to increased pattern matching costs, especially if a substring matches more than one pattern. We discuss the choice of patterns in Section 5.2.6.

### 5.2.2 Compression algorithm

The \texttt{COMRAD} algorithm compresses the input over multiple iterations, where at each iteration, one or more distinct substrings that are repeated at least \texttt{F} times are substituted with non-terminals. The algorithm terminates when no distinct substrings matching the pre-defined set of patterns remain with a frequency of at least \texttt{F}. Once all substitutions have been made, the output is encoded to produce the final compressed output. We now describe the algorithm in detail, beginning with the specification of the inputs and outputs of the algorithm.

#### Input

The \texttt{COMRAD} algorithm expects one or more DNA sequences as input. Let \texttt{S_0} be the set of \texttt{M} input sequences, where \texttt{S_0} = \{\texttt{S}_1^0, \texttt{S}_2^0, \ldots, \texttt{S}_M^0\}. Let \texttt{n_i^0} be the length of sequence \texttt{S}_i^0, where \texttt{1} ≤ \texttt{i} ≤ \texttt{M}. Then each sequence \texttt{S}_i^0 ∈ \texttt{S_0} is defined as \texttt{S}_i^0 = \texttt{s_1}s_2\ldots s_{n_i^0}, where \texttt{s_j} ∈ \texttt{Σ} for \texttt{1} ≤ \texttt{j} ≤ \texttt{n_i^0}. Apart from the sequences, the algorithm also takes the \texttt{L} and \texttt{F} parameters described in Section 5.2.1 as input. An example input sequence and the \texttt{F} and \texttt{L} parameters are illustrated in Figure 5.2.

The first iteration takes \texttt{S_0} as the input and substitutes repeated substrings to produce the output set of sequences \texttt{S_1} and a dictionary of substitution rules \texttt{R_1}.
Example 1 pattern $AB$:

$$aacgattgcacgattgctttacgattgcacgattgctg$$

$aBttBg$

$A \rightarrow acgattgc$

$B \rightarrow AA$

Example 2 pattern $x_1x_2B$:

$$aacgattgccggacgattgtcctagcctagggacgattgcgacgattgct$$

$aBttBgAt$

$A \rightarrow acgattgc$

$B \rightarrow AggA$

Example 3 pattern $Ax_1x_2x_3x_4x_5$:

$$aacgattgcctctcctagggacgattgctcctaggcctacgattgct$$

$aBggggBgctAt$

$A \rightarrow acgattgc$

$B \rightarrow AtcctA$

Example 4 pattern $x_1x_2x_3x_4x_5B$:

$$atcctagacgattgcggtcctagacgattgcctacgattgtct$$

$atBggtBctAt$

$A \rightarrow acgattgc$

$B \rightarrow cctagA$

Figure 5.1: Examples to illustrate the use of the patterns to detect and replace repeats in the subsequent iterations. The first line of each example contains the pattern being detected, and the second line contains the input sequence with the detected repeats underlined, followed by the final substituted sequence, and the dictionary of substitution rules.

<table>
<thead>
<tr>
<th>Input sequences ($S_0$)</th>
<th>Input parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0^1$: ctaccgtttat</td>
<td>$L = 4$</td>
</tr>
<tr>
<td>$S_0^2$: ataccgttttag</td>
<td>$F = 2$</td>
</tr>
<tr>
<td>$S_0^3$: ttaacctggttaa</td>
<td></td>
</tr>
<tr>
<td>$S_0^4$: gtaccaagtttat</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.2: The example input sequences and input parameters into COMRAD.
5.2. COMRAD COMPRESSION

**Frequency dictionary \( (D_1) \)**

- `aact : 2`  `ccgt : 2`  `gtaa : 1`  `tacc : 4`  `ttag : 1`
- `accg : 2`  `cggt : 2`  `gtac : 1`  `tggt : 2`  `ttat : 2`
- `actt : 2`  `ctac : 1`  `gtgt : 2`  `tgtt : 2`  `ttgg : 2`
- `atac : 1`  `cttg : 2`  `gtta : 4`  `ttaa : 1`

Figure 5.3: Frequency dictionary for distinct \( L \)-mers in the sequences of Figure 5.2.

The second iteration takes \( S_1 \) as the input and substitutes repeated substrings to produce the output set of sequences \( S_2 \) and a dictionary of substitution rules \( R_2 \). The input into the \( k \)th iteration is the set of sequences \( S_{k-1} \), and the output of the iteration is the set of sequences \( S_k \) and dictionary of substitution rules \( R_k \).

Each iteration of the COMRAD algorithm consists of two phases; in the first phase, the frequency counts of distinct substrings are determined, and in the second phase substitutions are made based on the frequencies of the substrings. In our discussion, we distinguish between the first iteration of the algorithm and the subsequent iterations of the algorithm, since the input into the first sequence is a series of nucleotide sequences, whereas the input into the subsequent iterations consists of sequences containing nucleotides as well as non-terminals.

**First iteration dictionary construction**

The dictionary \( D_1 \) is used to count the frequencies of all distinct \( L \)-mers in each sequence \( S_0^i \in S_0 \), where \( 1 \leq i \leq M \). Let \( s_{j,j+L} \) be a substring from a sequence, and let \( \overline{s}_{j,j+L} \) be its reverse complement. If \( s_{j,j+L} \) is in \( D_1 \), the count of the substring is incremented. If \( s_{j,j+L} \) is not in \( D_1 \), but \( \overline{s}_{j,j+L} \) is in \( D_1 \), then the count of the reverse complement entry is incremented. If neither the substring nor its reverse complement are in \( D_1 \), then \( s_{j,j+L} \) is added into \( D_1 \) with a count of one. This process is followed to count the frequencies of all distinct \( L \)-mers in the sequences of \( S_0 \). For the example set of sequences and parameters in Figure 5.2, the frequencies of the distinct \( L \)-mers are in Figure 5.3.

Recall from Section 5.2.1 that substrings with a frequency less than \( F \) are not eligible to be substituted by a non-terminal. Therefore it is unnecessary to keep such entries in \( D_1 \), so once all the \( L \)-mers are counted, a single pass through \( D_1 \) is used to remove entries with a frequency below \( F \). The distinct \( L \)-mers with frequency of at least \( F \) for the input sequences in Figure 5.2 are shown in Figure 5.4.

**First iteration substitution**

Given the disk-based nature of COMRAD, the substitution decisions made for each sequence are independent of the decisions made in the other sequences. In Section 5.1 the importance of ensuring that the algorithm makes the same substitutions for
identical repeats occurring across sequences was emphasised. Otherwise, in the next iteration, identical repeat regions will not be identified as being identical. We use the candidate selection algorithm proposed in the 5th modification in Section 5.1 to increase the likelihood of this property being satisfied.

Let $R_1$ be the set of substitution rules for the first iteration. Initially $R_1$ is empty. The sequences in $S_0$ are processed one sequence at a time. For each sequence $S_i^0 \in S_0$, a first pass is used to determine all frequent $L$-mers in the sequence (ie. has a frequency of at least $F$ in $D_1$). Then all frequent $L$-mers from the sequence are sorted from highest to lowest frequency. Then, starting from the most frequent $L$-mer until the least frequent $L$-mer, each $L$-mer $s$ that does not overlap with an existing substitution is replaced by a non-terminal. If $s$ does not overlap with an existing substitution, first $R_1$ is checked for the existence of substring $s$. If $s$ has been substituted before, the non-terminal $N$ associated with $s$ is retrieved from $R_1$. Otherwise, a new non-terminal $N$ is generated to represent $s$ and the rule $N \rightarrow s$ is added to $R_1$. The occurrence of $s$ is substituted with $N$. The substitutions continue until all non-overlapping frequent $L$-mers are substituted by non-terminals. The modified sequence $S_i^1$ of the input sequence $S_i^0$ is then written to disk.

The substitution algorithm ensures that the most frequent $L$-mers are substituted before less frequent $L$-mers. Since the same dictionary $D_1$ is used to make the substitution decisions, it is likely that the same decisions will be made in all sequences containing the same series of repeated $L$-mers. The substitution process for the input set of sequences in Figure 5.2 using the frequency dictionary from Figure 5.2.2 is depicted in Figure 5.5. The same repeated region is in sequence $S_1^0$ and $S_2^0$. Although the substitutions in sequences $S_1^0$ and $S_2^0$ are made independently, the repeat region has identical substitutions made with $A$ and $B$ non-terminals. As a result, in the next iteration, the substring $AgtB$ will be recognised as a repeat and substituted, removing the redundancy caused by the repeat $taccgtgta$. The output of the iteration is the set of sequences $S_1$ and the set of substitution rules $R_1$.

In the subsequent iterations, the input set of sequences consist of nucleotides and non-terminals. Since all $L$-mers with a frequency of at least $F$ are guaranteed to be covered by a non-terminal after the first iteration, in subsequent iterations, the only repeated substrings are those containing both terminal and non-terminal symbols. Therefore, patterns defined in Section 5.2.1 are used to identify repeats.
**Substitution** \((S_0, D_1) \rightarrow (S_1, R_1)\)

\[
S_0^1: \text{ctaccgtgttat} \quad \downarrow \quad S_0^2: \text{ataccggtttag} \quad \downarrow
\]

\[
tacc : 4 \quad \text{g} \quad \text{t} \quad \text{a} \quad \text{c} \quad \text{c} \quad \text{g} \quad \text{t} \quad \text{a} \\
\text{cttg} : 2 \quad \text{g} \quad \text{t} \quad \text{g} \quad \text{t} \quad \text{g} \quad \text{t} \quad \text{t} \quad \text{a} \quad \text{t} \\
\]

\[
S_1^1: \text{cAgtBt} \quad \downarrow \quad S_1^2: \text{aAgtBg} \quad \downarrow
\]

\[
S_0^3: \text{ttaacttggttata} \quad \downarrow \quad S_0^4: \text{gtaccaagttat} \quad \downarrow
\]

\[
taac : 4 \quad \text{g} \quad \text{g} \quad \text{t} \quad \text{a} \quad \text{a} \quad \text{c} \quad \text{c} \quad \text{a} \quad \text{a} \quad \text{g} \quad \text{t} \quad \text{a} \\
tttg : 2 \quad \text{t} \quad \text{g} \quad \text{g} \quad \text{t} \quad \text{t} \quad \text{a} \quad \text{t} \\
\]

\[
S_1^3: \text{tBttAtA} \quad \downarrow \quad S_1^4: \text{gAaaBt} \quad \downarrow
\]

\[
R_1: A \rightarrow \text{tacc}, \quad B \rightarrow \text{gtta}
\]

Figure 5.5: The substitution step of the first iteration for the input sequences \(S_0\) in Figure 5.2. Note that the substitutions in \(S_0^3\) are reverse complement substitutions (\(\text{tacc}\) is the reverse complement of \(\text{gtta}\), and \(\text{ggta}\) is the reverse complement of \(\text{tacc}\)). The output of the iteration is the set of modified input sequences \(S_1\), and the set of substitution rules \(R_1\).

Let \(k\) be the current iteration number, where \(2 \leq k \leq K + 1\). The largest iteration number is \(K + 1\), since an extra iteration is necessary to determine if all possible substitutions have been made. The input into iteration \(k\) is the output set of sequences from the previous iteration \(S_{k-1}\), the parameter \(F\) and the set of patterns \(P\). Similar to the first iteration, each subsequent iteration consists of two phases, the dictionary construction phase and the substitution phase.

**Subsequent iterations dictionary construction**

Let \(D_k\) be the frequency dictionary for the iteration. Let \(p\) be a substring in a sequence that matches one of the patterns in \(P\), and let \(\bar{p}\) be its reverse complement.\(^1\)

If \(p\) exists in \(D_k\), then its count is incremented. If \(p\) is not in \(D_k\), but \(\bar{p}\) exists in \(D_k\), then the count of \(\bar{p}\) is incremented. If \(p\) nor \(\bar{p}\) are in \(D_k\), then \(p\) is added to \(D_k\) with a count of one. This process is followed to count the frequencies of every substring \(p\) that matches a pattern in \(P\). After all sequences are read, the frequency dictionary \(D_k\) contains the counts of all substrings in \(S_{k-1}\) that match a pattern in \(P\).

Once again, the substrings in \(D_k\) with a frequency less than \(F\) are not eligible for substitution. Therefore, a single pass through \(D_k\) is used to remove these infrequent substrings. If the dictionary is empty after this step, then there were no substrings

---

\(^1\)The reverse complement of the patterns \(AXB\), \(XB\) and \(AX\) are \(\overline{BXA}\), \(\overline{BX}\) and \(\overline{XA}\), respectively.
CHAPTER 5. EFFICIENT GENOME STORAGE WITH COMRAD

Frequency dictionary ($D_2$)

\[
\begin{align*}
\text{AgtB} & : 2 \\
\text{BttA} & : 2
\end{align*}
\]

Figure 5.6: The second iteration frequency dictionary for distinct substrings matching the patterns in $P$ for the input sequences $S_1$ in Figure 5.5.

left to be replaced. Compression is terminated and the output consisting of the set of sequences in $S_k$ and the substitution rules $R = R_1 + \ldots + R_{k-1}$, are given to the substitution rule cleanup phase described in Section 5.2.2. The frequency dictionary of the second iteration for the sequences $S_1$ in Figure 5.5, is illustrated in Figure 5.6. Since all substrings in $D_2$ have a frequency of at least $F$, no entries were removed from this dictionary.

**Subsequent iterations substitution**

The substitution step of the subsequent iterations, is similar to that of the first iteration, except, instead of substituting frequent L-mers, frequent substrings matching patterns in $P$ are substituted. The same candidate selection algorithm as in the first iteration is applied for the reasons specified in Section 5.2.2.

Let $R_k$ be the set of substitution rules for iteration $k$, where $2 \leq k \leq K$. Initially, $R_k$ is empty. For each sequence $S_{k-1} \in S_k$, the frequent substrings in $S_{k-1}$ (with a frequency of at least $F$) are recorded. The substrings are then sorted in the most frequent to least frequent order. Beginning from the most frequent substring, each substring $s$ is substituted with a non-terminal if $s$ does not overlap with an existing substitution in the current iteration. If $s$ is to be substituted, $R_k$ is checked to see if $s$ has an associated non-terminal, and if one exists, then the non-terminal $N$ is retrieved from $R_k$. If $s$ is not in $R_k$, then a new non-terminal $N$ is generated and the rule $N \rightarrow s$ is added to $R_k$. Then substring $s$ is substituted by $N$. This process continues until all possible frequent substrings are substituted in sequence $S_{k-1}$ to produce the sequence $S_k$. The sequence $S_k$ is written to disk. The substitutions made to the sequences $S_1$ from Figure 5.5 using the frequency dictionary $D_2$ from Figure 5.6 is presented in Figure 5.7. The output of the iteration is the set of sequence $S_k$ and the set of substitution rules $R_k$.

**Output**

The process continues for $(K + 1)$ iterations, where in the $(K + 1)$th iteration for $K \geq 1$, the dictionary of substrings only contain substrings with a frequency less than $F$, as described in Section 5.2.2. An alternative terminating condition for the algorithm is to use a pre-defined number of iterations. This may be necessary to reduce the compression time if a sufficient compressed result is achieved in the earlier
5.2. COMRAD COMPRESSION

Substitution \((S_1, D_2) \rightarrow (S_2, R_2)\)

\[
\begin{align*}
S_1^1 &: cAgtBt \\
S_2^1 &: aAgtBg \\
S_3^1 &: tBtt\bar{A}a \\
S_4^1 &: gAaaBt \\
\downarrow & \downarrow & \downarrow & \downarrow \\
AgtB &: 2 & AgtB &: 2 & Btt\bar{A} &: 2 & AaaB &: 2 \\
\downarrow & \downarrow & \downarrow & \downarrow \\
S_1^2 &: cCt \\
S_2^2 &: aCg \\
S_3^2 &: tDa \\
S_4^2 &: gDt \\
\end{align*}
\]

\[
R_1 : C \rightarrow AgtB, D \rightarrow \bar{B}tt\bar{A}
\]

Figure 5.7: The substitution step of the second iteration for the input sequences \(S_1\) in Figure 5.5. Note that the substitution in \(S_4^1\) is a reverse complement substitution (\(\bar{B}tt\bar{A}\) is the reverse complement of \(AaaB\)). The set of modified sequences \(S_2\) and the set of substitution rules \(R_2\) form the output of the iteration.

Output \((S_K)\)  Substitution rules \((R)\)

\[
\begin{align*}
S_K^1 &: cCt & A &\rightarrow tacc \\
S_K^2 &: aCg & B &\rightarrow gtta \\
S_K^3 &: tDa & C &\rightarrow AgtB \\
S_K^4 &: gDt & D &\rightarrow \bar{B}tt\bar{A}
\end{align*}
\]

Figure 5.8: The output of COMRAD for the input set of sequences in Figure 5.2.

iterations of the algorithm. The output of the algorithm is the set of sequences \(S_K\) and the dictionary of substitution rules \(R = R_1 + R_2 + \ldots + R_K\). The sequences \(S_K\) and the substitution rules \(R\), are given to the second last step of the algorithm to be “cleaned up” before the final step. The final step encodes the sequences \(S_K\) and the dictionary \(R\) to produce the compressed output. The output for the set of sequences from Figure 5.2 is shown in Figure 5.8.

Dictionary cleanup

After the many iterations of substitutions, it is possible that certain non-terminals will appear in the output less than \(F\) times. The aim is to have every non-terminal appear at least \(F\) times. Therefore, the non-terminals no longer occurring at least \(F\) times are deemed to not be worth the encoding cost, so must be replaced with their original substrings. In the dictionary cleanup step, COMRAD replaces all non-terminals not occurring at least \(F\) times with their original substring. The substitutions occur recursively until all non-terminals have a frequency of at least \(F\). In a similar manner, in the second last step of Sequitur [Nevill-Manning and Witten, 1997b], the dictionary and the sequences are cleaned up.
The **COMRAD** algorithm uses a unique positive integer to represent each non-terminal. If the substitution rules are sorted in increasing rule identifier order, then just the right-hand side of the rules can be encoded and the left hand side of the rule can be inferred by the ordering to save space. One of the consequences of removing infrequent non-terminals is that the non-terminal identifiers will no longer be consecutive. To ensure that all remaining non-terminals are assigned consecutive non-terminal numbers, an extra pass through the set of sequences and the substitution rules is used to remap the non-terminal identifiers. The “cleaned up” set of sequences $S_K$ and the set of substitution rules $R$ are then given to the encoder to obtain the final compressed representation, as described next.

**Huffman coding**

In this final step, the set of sequences in $S_K$ and the set of substitution rules in $R$ are encoded to produce the final compressed output. We use Canonical Huffman coding, which was described in Section 3.2.5. Huffman coding was chosen for its speed and simplicity, although other encoding schemes may produce better compressed results. Huffman coding can encode a symbol in a number of bits close to the 0-order entropy, if most symbols have low probabilities. The nucleotide symbols have high probabilities compared to the less frequent non-terminal symbols. Therefore, instead of Huffman coding single nucleotides, we Huffman code $n$-mers, where $n$ varies from 1–6, to reduce the nucleotide probabilities to be close to that of the non-terminals. The Huffman-coded **COMRAD**-compressed sequences, substitution rules, and other information required for decompression are written to disk for storage.

The **COMRAD** decompression algorithm is discussed next.

**5.2.3 Decompression algorithm**

The decompression algorithm consists of two steps; the Huffman-coded sequences and substitution rule dictionary are first decoded, then the **COMRAD**-compressed sequences are expanded using the substitution rule dictionary to reproduce the original sequences. Below, we describe each step in detail.

**Huffman decoding**

Each encoded sequence and the encoded substitution rule dictionary are decoded using the Canonical Huffman decoding algorithm to reproduce the **COMRAD**-compressed collection. The sequences and the dictionary are written to disk as it may be too expensive to hold it in memory for the next step.
5.2. COMRAD COMPRESSION

Substitution rules \((R)\)

\[
\begin{align*}
A & \rightarrow \text{tacc} \\
B & \rightarrow \text{gtta} \\
C & \rightarrow \text{AgtB} \\
D & \rightarrow \text{BttA}
\end{align*}
\]

Decode \((S^1_K)\)

\[
S^1_K: \text{cCt} \quad \Rightarrow \quad \text{Expand C} \quad \Rightarrow \quad \text{Expand A (tacc)} \quad \text{Expand B}
\]

\[
(\text{ctaccgtgttat}) \quad \Leftarrow \quad (\text{taccgtgtta}) \quad \Leftarrow \quad (\text{gtta})
\]

Decode \((S^2_K)\)

\[
S^2_K: \text{aCg} \quad \Rightarrow \quad \text{Expand C} \quad \Rightarrow \quad \text{Expand A (tacc)} \quad \text{Expand B}
\]

\[
(\text{ataccgtgttag}) \quad \Leftarrow \quad (\text{taccgtgtta}) \quad \Leftarrow \quad (\text{gtta})
\]

Figure 5.9: Decompressing the first two sequences from the compressed collection in Figure 5.8. This version stores the standard substitution rule dictionary.

Non-terminal decoding

To decompress the COMRAD-compressed sequences, all substitution rules need to be in memory to have constant time access to each rule for fast decompression. We consider two alternatives for storing the substitution rules in memory. The first is to store the right-hand side of each rule. The non-terminal identifiers for each rule can be inferred from the order in which the rules are stored. Recall that the rules produced by COMRAD are hierarchical, where a non-terminal can represent substrings consisting of other non-terminals. When decoding a non-terminal, the dictionary is accessed to obtain the substring represented by this non-terminal. If the substring contains non-terminals, these are decompressed recursively until the substring consists of just terminal symbols. An example decoding using this method is shown in Figure 5.9 for two of the sequences in the set of compressed sequences in Figure 5.8. A disadvantage of this method is illustrated in Figure 5.9. The non-terminal \(C\) occurs in the sequence \(S^1_K\) and \(S^2_K\). Therefore, the substring associated with \(C\) needs to be recursively decompressed twice. While this may be a small cost in this example sequence, in large collections with many copies of long repeats, the reduced decompression speed as a result of recursive decompression will be evident.

The second alternative is to store the decompressed right-hand sides of the sub-
Substitution rules \((R)\)

\[
\begin{align*}
A & \rightarrow \text{tacc} \\
B & \rightarrow \text{gtta} \\
C & \rightarrow \text{taccgtgtta} \\
D & \rightarrow \text{taacttggtta}
\end{align*}
\]

Decode \((S^1_k)\)

\[
S^1_k: \text{cCt} \quad \Rightarrow \quad \text{Expand C} \\
(\text{ctaccgtgttat}) \quad \Leftarrow \quad (\text{taccgtgtta})
\]

Decode \((S^2_k)\)

\[
S^2_k: \text{aCg} \quad \Rightarrow \quad \text{Expand C} \\
(\text{ataccgtgttag}) \quad \Leftarrow \quad (\text{taccgtgtta})
\]

Figure 5.10: Decompressing the first two sequences from the compressed collection in Figure 5.8. This version stores the substitution rules with their right-hand sides decompressed.

Substitution rules. Then, when a non-terminal is decoded, the substring represented by the non-terminal can be accessed directly from the dictionary. An example of this process is shown in Figure 5.10 for the first two sequences in the set of compressed sequences in Figure 5.8. As seen in the example, the substring represented by non-terminal \(C\) can be directly accessed from the dictionary. The expanded dictionary could consume more memory than the set of hierarchical rules. But, since decompression is likely to be frequent, in COMRAD, we favour faster decompression speed over the memory cost, so we adopt the latter approach. Larsson and Moffat [1999] also adopted the latter approach for Re-pair.

Once the dictionary is stored in memory, decoding sequences is straightforward. One compressed sequence is read at a time, and each symbol in the sequence is decoded. If the symbol being decoded is a terminal, then the nucleotide is written to the output file on disk, where the decompressed sequence is stored. If the symbol being decoded is a non-terminal, then the substring of nucleotides represented by the symbol are retrieved from the dictionary and written to disk. This process continues until all the sequences are decompressed.

Since COMRAD preserves sequences boundaries, it is possible to decompress a selected sequence without decompressing the entire collection. This is an important feature, especially for large collections. Another advantage of COMRAD is that it can
be modified to retrieve substrings from compressed collections. The modifications required to implement this random access feature are discussed in Section 5.3. Next we discuss the algorithmic complexity of COMRAD.

5.2.4 Time and space complexity

Below we describe the complexity of each step of the COMRAD algorithm, although it is not trivial as the complexity depends on the repeat properties of the collection. Recall that the input to the algorithm consists of a set \( S_0 = \{ S_0^1, S_0^2, \ldots, S_0^M \} \) of \( M \) sequences, and \( N_0 = \sum_{i=1}^{M} n_0^i \) is the length of the collection, where \( n_0^i = |S_0^i| \). The analysis assumes that the dictionary is implemented as a hash table where items can be inserted and searched for in amortized \( O(1) \) time.

First iteration: In the frequency dictionary construction step, the algorithm uses a single pass through each sequence to calculate the frequencies of \( L \)-mers. Therefore, this step has an \( O(N_0) \) time complexity. The space complexity of the step is dependent on the number of distinct \( L \)-mers in the collection. In the worst case, all \( L \)-mers could be distinct so the space complexity is \( O(N_0) \).

In the substitution step of the first iteration, the worst case time complexity for substituting the \( L \)-mer repeats in each sequence \( S_0^i \) is \( O(n_0^i \log n_0^i + 2n_0^i) = O(n_0^i \log n_0^i) \). Two passes through the sequence are required to identify candidates for substitution, then to make the substitutions, hence the \( 2n_0^i \) cost. The \( O(n_0^i \log n_0^i) \) cost is to sort the possible substitutions from highest to lowest frequency (in the worst case, substrings starting at each of the \( n_0^i \) positions in the sequence can be eligible for substitution). Since only a single sequence at a time is read into memory, the space complexity of the substitution step is \( O(N_0 + n_0^i) \), where the \( O(N_0) \) cost is for storing the frequency dictionary, and the \( O(n_0^i) \) cost is for storing the longest sequence \( S_0^i \) in the collection in memory.

Subsequent iterations: The analysis for the subsequent iterations is not trivial, since the complexity depends on the length of the output sequences from the previous iteration, which in turn depends on the repeat properties of the collection. The frequency dictionary construction step is linear in the length of the input sequences into the iteration. At iteration \( k \) for \( k \geq 2 \), let the length of the input sequences be \( N_{k-1} = \sum_{i=1}^{M} n_{k-1}^i \), where each \( n_{k-1}^i \) is the number of symbols in the sequence \( S_{k-1}^i \). A single pass through the sequences in \( S_{k-1} \) is required to build the frequency dictionary for the iteration. For each position in a sequence, at most \( |P| \) comparisons are made to determine whether a substring starting at that position matches a pattern, and if so which pattern the substring matches to. Therefore, the time complexity of the frequency dictionary construction step is \( O(|P|N_{k-1}) = O(N_{k-1}) \), since \( |P| \) is constant. The space complexity of this step is \( O(N_{k-1}) \), since in the worst case, the substring starting at each position in each sequence could be distinct.
The substitution step of the \( k \)th iteration is similar to the substitution step of the first iteration. The worst case time complexity of substituting the repeated substrings in each sequence \( S'_{k-1} \) is \( O(n_{k-1}^i \log n_{k-1}^i) \), since at most \( n_{k-1}^i \) substrings will be eligible to be substituted, and they need to be sorted to determine the order of substitutions. The space complexity is \( O(N_{k-1} + n_{k-1}^i) \) for storing the frequency dictionary of substrings and the longest sequence in memory.

**Dictionary cleanup:** The dictionary cleanup step, where non-terminals with a frequency less than \( F \) are replaced by the substring that they represent, requires a linear pass through the final dictionary of substitution rules \( R \) and the compressed set of sequences \( S_K \). If there are \( r \) rules in \( R \) and the number of symbols in \( S_K \) are \( N_K \), then the time complexity of this step is \( O(r + N_K) \). All the rules in \( R \) need to be stored in memory to count the actual number of occurrences of each non-terminal and to revert substitutions for the infrequent non-terminals. To revert infrequent substitutions, a single sequence is read into memory at a time. Therefore, the space complexity of this step is \( O(r + n_K') \), where sequence \( S_K' \) is the longest sequence.

**Huffman coding:** The Huffman coding step expects the non-terminals and \( n \)-mers of nucleotides to be sorted according to their codeword lengths (based on the probabilities of their occurrence). Let \( r \) be the number of non-terminals in the dictionary and let \( n' \) be the number of distinct \( n \)-mers. In practice, \( n' \ll r \) since \( n \) and \( |\Sigma| \) are small. Huffman coding the substitution rule dictionary is linear in the number of rules \( r \), and the encoding of the sequences is linear in the length of the compressed sequences \( N_K \). Therefore, the time complexity of this step is \( O(r \log r + r + N_K) = O(r \log r + N_K) \). The Huffman coding step requires the codeword lengths of each dictionary entry to be in memory, therefore this step has a space complexity of \( O(r) \) assuming \( n' \ll r \). The substitution rules \( R \) nor the sequences need to be in memory as they are read from disk one symbol at a time.

**Decompression:** The Huffman decoding step of the decompression process is linear in the number of entries in the substitution rule dictionary and the number of symbols in the compressed sequences; \( O(r + N_K) \). The space complexity is \( O(r) \), as for the Huffman coding step. The time complexity of the decompression step is linear in the length of the original collection \( (O(N_0)) \); each terminal symbol is output as is, and the terminal symbols represented by each non-terminal symbol are output after a constant time lookup in the substitution rule dictionary. The number of symbols being output is equivalent to the number of symbols in the original input, hence the \( O(N_0) \) time complexity. The space usage of this step is for storing the expanded substitution rules, and the size depends on the repeat properties of the collection. If the average length of a repeat is \( \bar{n} \), then the space usage for this step is approximately \( r\bar{n} \). It is difficult to define the space complexity for this step with-
out knowledge of the model that describes the repeat properties of the collection. Overall, the decompression step has $O(r + N_K + N_0)$ time complexity.

We discussed the worst case complexities of COMRAD without assuming any knowledge of the repeat properties of the collection. In practice, for a collection with even a small amount of repetition, the space complexity of the first iteration will be smaller than $O(N)$. And the larger the repeat content, the lower the space usage in the first iteration. For the subsequent iterations, it is not trivial to express the time and space complexities in terms of the size of the input collection. The size of the input to each iteration depends on the substitutions made at the previous iterations, and this is based on the repeat content of the sequences, and the relative frequencies of each repeated substring with respect to the other repeats in each sequence.

It is possible to model the repeat content of a collection using appropriate probability distributions, but the assumptions made for such a model will not be general enough for the wide range of DNA sequences that could be compressed with COMRAD. Therefore we do not attempt to model the repetitions in the input collections.

In general, the time and space complexity of COMRAD depends on the repeat properties of the collection. The higher the repeat content, the lower the time and space usage of the algorithm. The longer the average repeat length, the more iterations that are required to compress the input collection. Finally, the larger the collection being compressed, the greater the time taken to compress and decompress.

In terms of the efficiency of the grammar generated by COMRAD, the aim is to generate the smallest grammar that can represent the input collection. This is otherwise known as the smallest grammar problem, which is an NP-hard problem [Charikar et al., 2005; Storer and Szymanski, 1982]. Most grammar-based compression algorithms are heuristics to solve this problem, and [Charikar et al., 2005] found upper and lower bounds for the approximation ratios for algorithms such as, LZ78, Sequitur, and Re-pair. We do not know the approximation ratios for COMRAD. The current substitution algorithm adopted by COMRAD is a greedy strategy and this may not be the best algorithm. Further improvements can be made to the substitution algorithm but we leave this as a future exercise.

Considering its ability to detect repetitions across multiple sequences in a collection, the COMRAD algorithm needs to be implemented carefully to ensure that large collections can be compressed. The implementation details are discussed in Appendix B. The current implementation is not the best possible implementation of the algorithm and throughout the discussion, we emphasised the aspects of the implementation that can be improved. Given that COMRAD aims to detect repeats globally for large collections, it is inevitable that the algorithm will be memory intensive. We have attempted in our implementation to reduce this memory use but improvements still need to be made as discussed in Section 5.2.7. Next we experimentally verify the performance of our COMRAD implementation, and compare it to the general-purpose and DNA compression algorithms analysed in Chapter 4.
Table 5.1: The collections and sequences in DATASET-REP and DATASET-SIN, their sizes (in Mbases) and the average number of times a distinct 15-mer is repeated, as denoted in Tables 4.2b and 4.2a is reported as the average repeat rate, respectively. The number of sequences in each collection of DATASET-REP is also reported.

5.2.5 Experimental evaluation

Test data and environment

Since COMRAD is most suitable for compressing DNA sequence collections containing repetitions, we used DATASET-REP of Chapter 4 as the test dataset. Unlike in Chapter 4, we did not substitute the additional nucleotides in alphabet $\Sigma$ with nucleotides from alphabet $\sigma$, since COMRAD is designed to compress sequences derived from $\Sigma$.

We also compressed the sequences in DATASET-SIN to evaluate the performance of COMRAD on single sequence compression. Since the level of exact repetitions within sequences is typically low, we do not expect COMRAD to produce good compression results for the sequences from this collection. Tables 5.1a and 5.1b contains a summary of the properties of each collection in DATASET-REP and sequence in DATASET-SIN.

The compression performance of COMRAD was evaluated in terms of the three performance criteria described in Section 4.2.1. The compressed size was measured as the number of bytes required to store the compressed sequences and the substitution rules. The compression and decompression speeds were measured as the number of seconds required to read the input from disk, produce the compressed and decompressed output, and write the output to disk, respectively. The peak memory used during the compression and decompression process was measured using massif.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.
5.2. COMRAD COMPRESSION

Table 5.2: COMRAD compression results for the collections in dataset-rep. The columns are: the collection name, compressed size in Mbytes, average number of bits per base used, compression time in seconds, approximate maximum memory used during compression in Mbytes, decompression time in seconds and approximate maximum memory used during decompression in Mbytes, respectively.

<table>
<thead>
<tr>
<th>Coll.</th>
<th>Compressed size (Mbytes)</th>
<th>Bits per base</th>
<th>Compress time (seconds)</th>
<th>Compress memory (Mbytes)</th>
<th>Decompress time (seconds)</th>
<th>Decompress memory (Mbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>1.02</td>
<td>1.16</td>
<td>15</td>
<td>85.60</td>
<td>≪1</td>
<td>22.82</td>
</tr>
<tr>
<td>Mito</td>
<td>5.50</td>
<td>1.82</td>
<td>63</td>
<td>277.55</td>
<td>2</td>
<td>24.81</td>
</tr>
<tr>
<td>Infl</td>
<td>5.79</td>
<td>0.43</td>
<td>111</td>
<td>190.34</td>
<td>2</td>
<td>35.86</td>
</tr>
<tr>
<td>Ecol</td>
<td>14.80</td>
<td>0.75</td>
<td>376</td>
<td>1,557.48</td>
<td>5</td>
<td>85.07</td>
</tr>
<tr>
<td>Spara</td>
<td>17.47</td>
<td>0.34</td>
<td>1,067</td>
<td>1,644.55</td>
<td>7</td>
<td>153.00</td>
</tr>
<tr>
<td>Scere</td>
<td>14.58</td>
<td>0.25</td>
<td>718</td>
<td>1,327.87</td>
<td>7</td>
<td>131.89</td>
</tr>
<tr>
<td>Athal</td>
<td>65.62</td>
<td>1.09</td>
<td>4,963</td>
<td>6,237.60</td>
<td>22</td>
<td>401.39</td>
</tr>
<tr>
<td>Sson</td>
<td>18.34</td>
<td>0.16</td>
<td>20,566</td>
<td>2,205.62</td>
<td>13</td>
<td>188.25</td>
</tr>
<tr>
<td>Bact</td>
<td>630.72</td>
<td>1.91</td>
<td>13,853</td>
<td>20,000.00</td>
<td>227</td>
<td>797.57</td>
</tr>
<tr>
<td>Hsap</td>
<td>2,075.68</td>
<td>1.44</td>
<td>21,256</td>
<td>15,365.78</td>
<td>843</td>
<td>21,240.34</td>
</tr>
</tbody>
</table>

Results for repetitive sequence collections

The COMRAD compression results for the collections in dataset-rep is reported in Table 5.2. All collections except for the Hsap collection were compressed using the parameters $L = 16$ and $F = 4$. The first iteration frequency dictionary of Bact could not be stored in memory using a hash table, so we used the representation that only counts up to 255 (Approach 2 of Appendix B). The Hsap collections could not be compressed feasibly using the 16-mers representation, even with the truncated counting (18+ hours were required for the substitution step of the first iteration). Therefore, it was compressed using a version with the truncated counting and $L = 15$. The frequency threshold was kept at $F = 4$.

Overall, the collections with the higher average repetition rates, Infl, Scere, Spara, and Sson, achieved the best compression results, as shown by the lower average number of bits used to encode a base, compared to the remaining collections. The collection with the lowest average repetition rate, Mito, was not compressed well, but still achieved better compression than if the na"ive encoding was used (4 bpb due to the use of the extended DNA alphabet). The collections, Hemo, Ecol and Athal contained some repetition, and COMRAD was able to compress these to a reasonable level. We did not expect promising results for the Bact collection, given the genomes are from different species, but some compression was still achieved. The Hsap collection was not compressed as expected but we can expect that as more genomes are added to the collection, the compression will improve. For the Bact and Hsap collections, the truncated frequency counting in the first iteration may also have had a small impact on the compression results.

The number of iterations required by COMRAD to compress the collections were
also recorded. Overall, COMRAD required more iterations for larger collections than it did for smaller ones. The collections Hemo, Mito and Inf1 required around 10–11 iterations, while the remaining collections all required around 15–16 iterations. In general, the number of iterations depend on the length of the longest repeat in a collection. The Hemo, Mito and Inf1 collections contained short sequences, therefore the length of the longest repeat is also short. The remaining collections consisted of much longer sequences, hence the repeats are likely to be longer.

In terms of the compression time, COMRAD is fast to compress smaller collections, but slower for larger collections. Longer compression times can be expected for larger collections, since the algorithm is disk-based, so more data needs to be read from and written to the disk at every iteration. Although long compression times are undesirable, to handle global repeat detection across large collections, a disk-based algorithm is necessary. As shown in Chapter 4, in-memory algorithms such as Re-pair and Sequitur require that a large collection be compressed in blocks due to insufficient main memory being available to compress the entire collection. Larger collections also have larger processing times, in terms of the time taken to count substring frequencies and for substitutions, since the worst case complexities of these steps are proportional to the collection length. COMRAD compression speeds are also slower for less-repetitive collections, since such collections tend to contain many distinct repeats, which need to be managed during frequency counting. The decompression speed of COMRAD is much faster than its compression speed, and tends to be linear in the size of the collections. This is particularly advantageous, since decompression tends to be more frequent than compression.

COMRAD requires a significant amount of memory to store the distinct \( L \)-mers at the first iteration dictionary construction step. The memory use can be limited by using the more memory-efficient version that is optimised to store 15-mers using at most 5 Gbytes (Appendix B). In general, the maximum memory used during compression is around 2–3 times the input collection size for larger collections. Clearly this does not scale for compressing collections containing many gigabases or terabases. Methods for reducing memory usage are discussed in Section 5.2.7. On the other hand, during decompression, the memory usage is significantly less. Even after expanding the substitution rules, the memory occupied is less space than storing the original collection in memory. Also, the more repetitive the collection, the lower the memory used during decompression.

When comparing the COMRAD compression results to the best performing general-purpose and DNA compression algorithms from Chapter 4, COMRAD does not achieve better results than 7-Zip, Sequitur, dna-x and XMCompress, as shown in Figure 5.11. COMRAD out-performs Re-pair for the less repetitive collections, such as Hemo, Mito and Bact. The lower compression performance is due to the encoding techniques used by COMRAD not being optimised, whereas the authors of these algorithms have spent considerable efforts to optimise the encoding techniques used for
5.2. **COMRAD COMPRESSION**

Figure 5.11: Comparison of the compressed sizes (in bpb) for DATASET-REP.

their implementations. Optimising the COMRAD encoding techniques will be explored in future research. Also, the collections in Chapter 4 contain sequences from alphabet $\sigma$ instead of alphabet $\Sigma$, but the impact of this difference on the compressed size is unlikely to be significant as the additional nucleotides in $\Sigma$ occur rarely.

In terms of the compression speed, we did not expect COMRAD to be competitive with the other compressors, since it is a disk-based method, while the other compressors are memory-based. Observing the results, COMRAD tends to be faster to compress larger collections compared to the DNA compressors, XMCompress and dna-x. DNA compressors are research implementations, and tend not to be optimised for speed, so these results can be expected. Sequitur is consistently faster than COMRAD, while 7-Zip has similar speeds to COMRAD. Both these algorithms are adaptive compressors, hence we can expect them to be faster than COMRAD. Re-pair, the algorithm that is conceptually closest to COMRAD, tends to be faster to compress the highly repetitive collections, while being much slower for non-repetitive collections. This may be due to the fact that Re-pair only substitutes di-mers occurring at least twice, which requires more iterations. COMRAD, on the other hand, requires less iterations, since longer substrings with a higher frequency threshold are substituted.

In terms of the decompression speed, we expect COMRAD to be competitive with other tools, and the results confirm this hypothesis. Re-pair and dna-x tends to decompress slower than COMRAD, while 7-Zip is faster to decompress smaller collections but slower for larger collections, compared to COMRAD. As a statistical compressor, XMCompress has significantly slower decompression speeds than COMRAD.

Since COMRAD detects repeats globally with longer minimum repeat lengths, we
expect COMRAD to be less memory-efficient compared to other compressors. Surprisingly, 7-Zip, Sequitur, Re-pair and XMCompress, tend to be more memory intensive than COMRAD. The only tool that was consistently more memory-efficient compared to COMRAD was dna-x.

Due to the lack of publicly available large repetitive collections of DNA sequences, we generated some artificial sequences to determine the performance of COMRAD for compressing very large collections of DNA sequences. The model used to generate the artificial sequences is loosely based on the variations observed between human individuals. The algorithm takes as input, a DNA sequence, probability of a point mutation occurring, probability of an indel occurring, and an indel length, which are used to mutate the original sequence to produce two child sequences. Each child sequence is then mutated to produce two more child sequences, and so on. We used this algorithm to construct 127 variants of the human chromosome 1 (127xHsapChr1), and 1023 variants of human chromosome 20 (1023xHsapChr20), using a mutation probability of 1 in 10,000 bases being substituted by a random base, an indel probability of 1 in 10,000,000 and an indel length of 10,000 bases. Although the generated sequences are not representative of actual human genomes, it allows us to evaluate whether COMRAD is able to compress larger collections.

The 127xHsapChr1 collection was originally 31,411.70 Mbases in length, and it was compressed by COMRAD to 282.56 Mbytes (0.07 bpb) in 22 iterations. The 1023xHsapChr20 collection was originally 63,841.82 Mbases in length, and was compressed to 291.07 Mbytes (0.04 bpb). Although COMRAD achieved excellent compression results for these two collections, the compression time was around 48 hours per collection, which is significant. Nevertheless, the results show that COMRAD is able to compress very large sets of sequences, provided they are highly repetitive.

To evaluate the variation in compression when the mutation and indel rates are varied, we performed another experiment. We artificially generated 63 variants of the human chromosome 22 sequence using the above method, with varying mutation and indel rates. All sequences except for the 32 leaf sequences were discarded to ensure that each sequence is distantly related to the other sequences in the collection. The results for the experiment are shown on Table 5.3. As expected, when the mutation and indel rates are reduced, compression improves. The initial 10-fold reduction in the mutation and indel rate results in a significant reduction in the compressed size, from 1.26 bpb to 0.42 bpb. Each subsequent 10-fold reduction in the mutation and indel rate results in diminishing reductions in the compressed size. The 1 in 100 mutation rate is a very high rate of mutation that would not be observed in a collection of sequences from the same species but the compression achieved by COMRAD is still significant. Overall, the results show that the compression results are dependent on the mutation and indel rates that are observed in a collection of sequences, and even in the presence of high mutation rates, if the sequences in the collection are related, COMRAD can detect the repetition and compress the collection.
5.2. **COMRAD COMPRESSION**

<table>
<thead>
<tr>
<th>Collection name</th>
<th>Mutation rate</th>
<th>Indel rate</th>
<th>Bits per base</th>
</tr>
</thead>
<tbody>
<tr>
<td>32xHsChr22 set 1</td>
<td>1 in 100</td>
<td>1 in 100,000</td>
<td>1.2538</td>
</tr>
<tr>
<td>32xHsChr22 set 2</td>
<td>1 in 1,000</td>
<td>1 in 1,000,000</td>
<td>0.4205</td>
</tr>
<tr>
<td>32xHsChr22 set 3</td>
<td>1 in 10,000</td>
<td>1 in 10,000,000</td>
<td>0.1272</td>
</tr>
<tr>
<td>32xHsChr22 set 4</td>
<td>1 in 100,000</td>
<td>1 in 100,000,000</td>
<td>0.0784</td>
</tr>
</tbody>
</table>

Table 5.3: **COMRAD** compression results for collections containing varying mutation rates. The columns are the collection name, single point mutation rate, indel rate (an indel has a length of 10,000), and the average number of bits used to encode a base, respectively. Note that the rates are a parameter to the model of evolution and the actual level of difference between each of the sequences in a collection is dependent on the evolutionary relationship between a pair of sequences.

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (seconds)</th>
<th>Comp. memory (Mbytes)</th>
<th>Decomp. time (seconds)</th>
<th>Decomp. memory (Mbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EcoliK12</td>
<td>1.10</td>
<td>1.99</td>
<td>18</td>
<td>164.26</td>
<td>≪1</td>
<td>47.06</td>
</tr>
<tr>
<td>ScereRef</td>
<td>2.79</td>
<td>1.93</td>
<td>50</td>
<td>405.47</td>
<td>1</td>
<td>82.05</td>
</tr>
<tr>
<td>PfalcRef</td>
<td>4.81</td>
<td>1.73</td>
<td>77</td>
<td>405.47</td>
<td>1</td>
<td>40.27</td>
</tr>
<tr>
<td>CelegRef</td>
<td>23.95</td>
<td>1.93</td>
<td>595</td>
<td>1,557.47</td>
<td>8</td>
<td>158.85</td>
</tr>
<tr>
<td>AthalRef</td>
<td>27.68</td>
<td>1.95</td>
<td>673</td>
<td>3,093.47</td>
<td>10</td>
<td>183.56</td>
</tr>
<tr>
<td>DmelRef</td>
<td>33.57</td>
<td>1.67</td>
<td>1,289</td>
<td>3,093.47</td>
<td>12</td>
<td>196.22</td>
</tr>
<tr>
<td>GgalRef</td>
<td>240.12</td>
<td>1.95</td>
<td>18,301</td>
<td>12,317.60</td>
<td>88</td>
<td>1,003,30</td>
</tr>
<tr>
<td>HsapRef</td>
<td>665.18</td>
<td>1.80</td>
<td>33,305</td>
<td>12,309.48</td>
<td>250</td>
<td>1,344.48</td>
</tr>
</tbody>
</table>

Table 5.4: **COMRAD** compression results for compressing the sequences in dataset-sin. The columns are, the sequence name, compressed size in Mbytes, average number of bits per base used, compression time in seconds, approximate maximum memory used during compression in Mbytes, decompression time in seconds and approximate maximum memory used during decompression in Mbytes, respectively.

**COMRAD** is able to compress repetitive multi-sequence collections of DNA without pre-processing the data and produce reasonable compression results within reasonable time limits. The algorithm is fast to decompress, making it suitable for applications where decompression is more frequent than compression. The algorithm also permits decompression of individual sequences without decompressing the entire collection. The downside of **COMRAD** is its high memory use.

**Results for single sequences**

Next we analyse the performance of **COMRAD** for compressing the single sequences in dataset-sin, the results of which are in Table 5.4. As stated earlier, **COMRAD** is most suitable for compressing repetitive collections, so we do not expect to observe good compression results for single sequences, which are unlikely to be repetitive.

**COMRAD** was able to compress all of the sequences to a size smaller than what could be achieved by a naïve encoding. The results are comparable to the best
CHAPTER 5. EFFICIENT GENOME STORAGE WITH COMRAD

Figure 5.12: Comparison of the compressed sizes (in bpb) for DATASET-SIN.

general-purpose compressors from Section 4.3.2, as shown in Figure 5.12. Sequitur, 7-Zip and ppmd mostly out-perform COMRAD for the larger sequences, while COMRAD produced better compression results for the two smallest sequences. It should be noted that all sequences except for the two smallest sequences contain nucleotides from alphabet $\Sigma$, which may be why COMRAD was out-performed by 7-Zip, Sequitur and ppmd for the larger sequences, since these tools compressed the versions of these sequences containing only nucleotides from alphabet $\sigma$. Re-pair, the algorithm that is conceptually closest to COMRAD, produced the worst compression results for the sequences in DATASET-SIN. This shows that substituting di-mers instead of longer substrings can result in too many substitution rules, which contribute significantly to the encoding costs. Hence, our choice of substituting longer substrings is justified.

The DNA compression algorithms dna-x and XMCompress, out-performed COMRAD for all sequences. We expected XMCompress to out-perform COMRAD, since the algorithm detects approximate repetitions, and this type of repeat is more common within a sequence than exact repeats. However, the better dna-x results were unexpected. Although both COMRAD and dna-x substitute long substrings, COMRAD makes the substitution decisions in a more principled manner compared to dna-x, which greedily chooses the longest possible substitution starting at a given position. The most likely reason behind the COMRAD compression results is the fact that an explicit codebook is needed, which contributes significantly to the compression cost, especially when compressing non-repetitive sequences, while dna-x does not store an explicit codebook. This suggests that more principled substitution decisions may not be necessary for single-sequence compression to achieve good results.
5.2. COMRAD COMPRESSION

In terms of the compression speed, \texttt{COMRAD} is slower than the general-purpose compression tools analysed in Chapter 4. Compared to the DNA compressors, \texttt{COMRAD} was significantly faster to compress than \texttt{XMCompress}, while \texttt{dna-x} was faster for smaller sequences, and significantly slower than \texttt{COMRAD} for the two largest sequences. These are very similar results to what was observed for \texttt{DATASET-REP}. In terms of the decompression speed, both \texttt{dna-x} and \texttt{XMCompress} were slower than \texttt{COMRAD}. Although \texttt{COMRAD} does not produce better compression results compared to the DNA compressors, it has better compression and decompression speeds. In terms of memory use, \texttt{COMRAD} tends to require more memory than \texttt{7-Zip}, \texttt{Sequitur} and \texttt{ppmd}, especially for large sequences. However, \texttt{Re-pair} occupied more memory than \texttt{COMRAD}, since \texttt{Re-pair} stores the sequence and substitution rules in memory.

Overall, \texttt{COMRAD} performs as expected for single-sequence compression and produced no worse results than general-purpose compressors. Although DNA compressors produce better results than \texttt{COMRAD}, they cannot scale for compressing large sequences. Next, we discuss the effects of altering the algorithm parameters.

5.2.6 Effects of changing parameters

Now we discuss the affects on compression when the two parameters $L$ and $F$ are changed. In the experiments above, we used either $L = 16$ or $L = 15$ for implementation convenience and due to the theoretical reasons described in Appendix B. We used $F = 4$ as the frequency threshold, since this was a value established by the authors of \texttt{XRAY} \cite{Cannane2002}. We now use the \texttt{Hemo} and \texttt{Scere} collections from \texttt{DATASET-REP} to observe the effects of changing the $F$ and $L$ parameters in practice. The \texttt{Hemo} collection was chosen due to its average repeat properties. The collection is small, so repeats are likely to be short. The \texttt{Scere} collection is much more repetitive and larger, so we can expect the collection to have longer repeats. We first analyse the results of changing the frequency threshold $F$, then we analyse the $L$ parameter. Finally, we discuss the pattern set $P$.

Changing the frequency

We alter the values of the frequency threshold $F$ between 2 and 50 for the two collections \texttt{Hemo} and \texttt{Scere}. The starting substring length was kept at a constant $L = 8$. We expect that the higher the value of $F$, the lower the compression, because less of the repeats will occur at a higher frequency, reducing the number of substrings that are eligible for substitution at each iteration. We also expect that for very low frequencies, such as $F = 2$, the encoding cost of introducing a new non-terminal will be higher than the cost of not introducing the non-terminal. We also test the claim by the authors of \texttt{RAY} \cite{Cannane2001}, that $F = 4$ is the threshold at which introducing a new non-terminal is beneficial.

The results are presented in Figure 5.13. The x-axis contains the frequency
threshold used, and the y-axis contains the compressed size in terms of bits per base. For the *Hemo* collection, as expected, the compressed size increased as $F$ is increased from $F = 3$ onwards. Also at $F = 2$, introducing a new non-terminal is more expensive, hence the larger compressed size. For the *Hemo* collection, the best compression result is achieved when $F = 3$ if $L$ is restricted to a length of 8.

The results are somewhat inconsistent with our expectations for the *Scere* collection. Overall, as the frequency threshold increases, the compressed size also increases, as predicted. However, unlike the *Hemo* collection, an $F = 2$ threshold gives the lowest compression result at $L = 8$. After $F = 39$, the compressed size is approximately 2 bpb. This is because the *Scere* collection contains 39 individual genomes from the *S. cerevisiae* species, and some of the substrings are shared among all the sequences, with exactly one copy occurring in each genome. For $F > 39$, these substrings are no longer detected. It is worth noting that COMRAD defaults to compressing the collection to approximately 2 bpb if insufficient repeats are detected. This behaviour is not evident in the *Hemo* collection, since there are over 15,000 sequences in the collection and the $F$ value needs to be increased to a much larger value before detecting the point at which compression defaults to 2 bpb.

In general, smaller frequency threshold values lead to better compression, provided that the value is not too small, such that the cost of introducing a non-terminal is higher than encoding the repeated substring. Although this experiment did not confirm our hypothesis regarding $F = 4$, it showed that a threshold of $F = 4$ still
5.2. COMRAD COMPRESSION

Figure 5.14: The change in the compressed size of the Hemo and Scere collections when the initial substring length parameter $L$ is changed in COMRAD. The x-axis is the initial length $L$, and the y-axis is the compressed size measured in bits per base. Frequency threshold was restricted to $F = 4$.

provides good results. Given that the best $F$ value to be used seems to change for each collections, it may be worth exploring heuristics for choosing the best value.

**Changing the initial length**

Next we alter the initial substring length $L$ between 2 and 30 to observe the effect on the compressed size. We expect that for small values of $L$, most repeats in the input collection will be detected. But the encoding cost is high for longer repeats, since many non-terminals are required to represent a long repeat. Also, for small values of $L$, it is likely that an $L$-mer will occur in a collection at least $F$ times by chance, even though it is not part of a repeat, hence we cannot expect good compression results. We assume a small $L$ value ranges from 2 to 10. For larger $L$ values, the probability that an $L$-mer will occur at least $F$ times by chance is low, therefore $L$ length repeats are likely to be legitimate repeats. The higher the $L$ value, the better the compression result, since the repeat can be represented using fewer non-terminals, hence the encoding costs are lower. We expect this behaviour to be observed for $L$ values from 10 to 20. However, if $L$ becomes too large, then most repeats are too short to be detected, so the compressed size increases. This may be the case for $L$ values above 20, but for collections with many long repeats, $L$ could be much larger before compression performance degrades.

We present the results of varying the $L$ value from 2 to 30 in Figure 5.14. The x-axis contains the varying $L$ values, and the y-axis contains the compressed size of each collection in bits per base. As predicted, for both collections, the compressed
size is higher when the $L$ value is small, specifically less than 5. The results are worse for the Hemo collection than for the Scere collection. After $L = 5$, for both collections, the compressed size tends to decrease as $L$ increases. The values $L = 18$ and $L = 24$ produced the smallest compressed size for Hemo and Scere collections, respectively. This confirms our hypothesis that slightly larger values of $L$ provide the better compression results. After the minima, the compressed size increases in slight increments as expected for both collections as the $L$ value is increased.

Our default $L = 16$ value is close but less than the value recommended by the experiment for the Hemo and Scere collections. However, this has a negligible effect on the compressed size. Also a value of $L = 16$ was chosen for implementation efficiency, and it is an acceptable trade-off to produce a negligibly worse compression result to significantly improve the compression speed and memory usage. Therefore, we will continue to use $L = 16$ as our default initial substring length.

**Changing patterns**

The patterns we defined in Section 5.1 for identifying repeats in the subsequent iterations of COMRAD can be generalised as follows; $A x^t x^t B$ and $A x^{t'} B$, where $0 < t < L$ and $0 \leq t' < L$. For a length $L$, there are $3L - 2$ patterns of this form. In general, the more patterns are defined, the greater the number of repeats that can be detected. However, to find repeated substrings matching a pattern, for each non-terminal, at most all of the patterns in the pattern set $P$ need to be checked to see if there is a repeated substring that includes this non-terminal. The larger the number of patterns in $P$, the greater the cost of detecting repeated substrings. Therefore, as more patterns are introduced, the pattern matching costs increase, reducing the compression speed and increasing the memory costs of storing the distinct substrings matching the patterns. Therefore, the number of patterns should be limited to achieve a good trade-off between compression speed and compressed size.

We acknowledge that the patterns chosen in Section 5.1 for COMRAD may not be the best, especially since they ignore short nucleotide substrings that are on either side of non-terminals. However, we aim to detect long repeats using fewer iterations as possible to reduce the compression speed. Therefore, we chose patterns that favour detecting long repeats over short ones. The Huffman coding step will encode any short nucleotide repeats efficiently in the final step of the algorithm.

Next, we discuss the suitability of COMRAD for compressing large collections of DNA sequences, then suggest some further improvements that can be made to improve the efficiency of the algorithm.
5.2. COMRAD COMPRESSION

5.2.7 Discussion

The primary aim of this research was to test the feasibility of detecting repetitions globally for large repetitive DNA sequence collections, without pre-processing the input. COMRAD can compress a collection of any number of sequences containing the nucleotides from the extended DNA alphabet. There is no overhead associated with concatenating multiple sequences into a single sequence or substituting nucleotides from the extended alphabet with nucleotides from the standard alphabet, as was required for the algorithms we analysed in Chapter 4. COMRAD accepts input in FASTA format, which is a common format used to represent DNA sequences.

COMRAD makes no assumptions about the input being compressed. However, only collections containing repetitions will be satisfactorily compressed. Even for collections with low repetition, COMRAD is able to achieve better compression than a naïve encoding, as was shown by the experimental results.

The main advantage of COMRAD is its ability to detect repetitions occurring across multiple sequences, which is not a feature available in many of the tools we analysed in Chapter 4. Tools such as GenCompress and XMCompress permit a sequence to be compressed with respect to another sequence. However, this type of compression is only effective for collections containing sequences from the same species, as we show in Chapter 6, with our own relative compressor. COMRAD makes no such assumptions about the input collection, and can compress sequences from different species, provided the sequences have some similarity. The disk-based approach to detecting repetitions across multiple sequences allows COMRAD to compress collections larger than those that can be compressed with existing in-memory algorithms.

In Section 5.2.5, we compressed two artificial collections containing more bases than the memory available in our test environment. The general-purpose and DNA compression tools can compress large collections of sequences in blocks, but they are unable to detect the repeats that occur across the blocks. As a result, they produce sub-standard compression results for multi-gigabase collections.

We have shown that a semi-static dictionary compressor can feasibly detect global repetitions even for multi-gigabase collections, provided the collections are highly repetitive. For non-repetitive collections and collections that are in the terabase range, the current algorithm is not feasible due to the large memory usage for the frequency dictionaries, especially during the first iteration. Even for the collections consisting of a few gigabases, like Bact and Hsap, the memory use was significant. For larger collections, a fully disk-based algorithm may be necessary. Since most steps of the algorithm can be executed independently for each sequence in the collection, the algorithm can be parallelised to improve the compression speed.

While the substitutions can be conducted independently in a parallel variant of COMRAD, a frequency dictionary for the entire collection is required at each iteration. An alternative could be to divide the input into blocks and construct separate
frequency dictionaries for each block. Then the dictionaries for each block can be
merged to form a single frequency dictionary using a multi-way merge algorithm.
The memory use of this approach is restricted to the memory use for constructing
the frequency dictionaries for a single block. During the multi-way merge, substrings
with a frequency less than the threshold can be removed, so the final frequency dic-
tionary will have a smaller size. We still cannot guarantee that the merged frequency
dictionary will fit in memory, so additional improvements are necessary.

One way to ensure that the frequency dictionary can be held in memory is
to restrict the number of distinct $L$-mers or pattern substrings that are added to
the dictionary. It is possible to limit the number to a predetermined constant $Y$,
chosen based on the memory available and the level of compression desired. Then
the $Y$-most frequent substrings can be added to the dictionary using an algorithm
similar to that described by Cormode and Hadjieleftheriou [2009]. Another option
is to adopt a sampling approach where only 1 in $p$ substrings are used to construct
the dictionary. Although the frequency counts will not be accurate, the frequently
occurring substrings are likely to be encountered multiple times, hence the relative
differences between the frequencies are likely to be preserved. The compression
results with these modifications may not be as good as the results of the unrestricted
algorithm. But if the memory use and run time can be reduced with a small impact
on compression, then it is worth considering these alternatives.

Although the memory usage during decompression is unlikely to be more than
the memory usage during the first iteration of compression, it can still be significant
for large collections (the Hsap collection in the experimental results). The majority
of the memory used during decompression is to store the right-hand sides of the
substitution rules. Currently, the substrings are stored using one byte per nucleotide.
Instead, each nucleotide can be stored using 4 bpb. Another option is to store
encoded nucleotide substrings, such as a substring that is arithmetic encoded. This
requires each substring to be decoded when it is accessed, making the decompression
step slower. Also, if lowering memory use is important, then the substitution rules
can be stored in memory without expanding the right-hand sides.

According to our experimental results, COMRAD was unable to produce better com-
pression results than the best compressors from Chapter 4, namely 7-Zip, Sequitur,
Re-pair, dna-x and XMCompress. The exceptional results by most of these tools
were as a result of concatenating the sequences in a collection to compress as a sin-
gle sequence so that the repetitions are visible to the tool. Had the sequences being
compressed individually, the compression results would not have been as promising.
Given that COMRAD is able to detect repeats globally, it would have been able to
detect the same repeats as the other tools. The weaker compression performance
of COMRAD can be attributed to the weak encoding techniques used. We did not ex-
plode better encoding techniques, while the authors of some of the algorithms listed
above, particularly Sequitur, Re-pair and dna-x have explored various encoding
5.2. **COMRAD COMPRESSION**

Current techniques. We believe that the compression performance of COMRAD can be further improved by using better encoding techniques.

Currently, Huffman coding is used to encode groups of terminal and non-terminal symbols. An alternative is to use an $n$th order arithmetic coder for the groups of terminal symbols, and an integer coder for the non-terminals. Golomb coding is particularly suitable for encoding non-terminals, since the highest non-terminal identifier is known before encoding begins, so a suitable divisor can be chosen. The most space consuming component of the compressed output tends to be the encoded dictionary of substitution rules. The techniques used by Larsson and Moffat [1999] in Re-pair that encodes substitution rules by exploiting the fact that non-terminals belong to generations due to the hierarchical nature of the rules, can be adopted for COMRAD, given the similarity in the rule hierarchy of the two algorithms.

The current COMRAD implementation does not encode FASTA headers. One approach to encode these headers is to store the headers in a separate file and compress using an existing text compression algorithm such as gzip. However, this approach is not suitable for individual sequence decompression, as the entire header file must be decompressed to access the relevant header. Instead, using a method such as Huffman coding with an index for fast access to individual headers is more appropriate. For collections consisting of a few large sequences, the encoding mechanism chosen for FASTA headers will not affect the final compressed size. However, for collections of many small sequences, the compressed FASTA header may require more space than the compressed sequences themselves, especially if there are no common substrings in the headers. In such cases, the FASTA header compression may need to be customized based on the properties of the headers.

An improvement to the hash calculation for the dictionary can also be made. Currently, when counting frequencies using a hash table, separate hash values are calculated for substrings beginning at positions $j$, $j + 1$, $j + 2$ and so on. Instead of calculating individual hashes, a rolling hash function can be used, where the hash value calculated at position $j$ can be updated to obtain the hash value for the substring beginning at position $j + 1$ to reduce the cost of calculating hashes.

Another feature that is useful for handling large collections, especially if COMRAD is to be a fully disk-based algorithm, is the ability to add a sequence to an existing compressed collection. Currently, it is possible to add a new sequence to an existing collection by using the existing dictionary of substitution rules. However, if multiple new sequences are added, then new repeats are introduced into the collection, and it may be necessary to detect these repeats and add them to the dictionary. We believe this is an interesting problem that is worth exploring in future research.

A feature of significance in COMRAD is its ability to trivially extract individual sequences without decompressing the entire collection, or the entire file containing the sequence of interest. For the tools we analysed in Chapter 4, due to the need to concatenate the sequences in a collection into a single sequence, individual sequence
extraction is impossible without decompressing the entire collection. Not only does COMRAD enable extraction of individual sequences, it also enables access to substrings without decompressing the entire collection. Some modifications can be made to the compressed output of COMRAD to enable the random access feature. We discuss these modifications and the random access algorithm in the next section.

5.3 The display() query

For certain applications, it is not sufficient to just compress a large collection for efficient storage. The information in the collection may also need to be accessed regularly. Most standard compression algorithms require that the entire collection be decompressed even to access a short substring, which is a significant overhead. As we described in Section 3.4, self-indexes implement queries to access and search the compressed collection. While we did not build a complete self-index on a COMRAD-compressed collection, we implemented the random access query to retrieve substrings from a COMRAD-compressed collection. This is otherwise known as the display() query and in this section, we describe how to implement this query for COMRAD. Recall that display(i,s,e) retrieves the substring $S^o_s[e]$.

First we describe the random access algorithm for a COMRAD-compressed collection. Then we discuss the algorithmic complexity for a query, along with a discussion of how to modify the output of the compression algorithm to implement the query. Finally, we present results that compare the query performance of COMRAD to other algorithms that implement the same query, such as LZ-End and RLCSA.

5.3.1 Algorithm

Recall that COMRAD outputs a set of compressed sequences $S_K$ and a set of substitution rules $R$. We rename $S_K$ to $S_c$ to indicate that it is the compressed set of sequences, and the original set of sequences are renamed to $S_o$.

First we attempt to retrieve a single nucleotide from a compressed sequence. The difficult aspect of the algorithm is determining where nucleotide $S^i_s[j]$ occurs in the compressed sequence $S^i_c$. The nucleotide can either be within a non-terminal or it can be an uncompressed nucleotide in $S^i_o$. Figure 5.15 shows sequence $S^i_s$ from the example in Figure 5.8 and the corresponding compressed sequence $S^i_c$, along with the substitution rules relevant to this sequence. The substring taccgtgtta is compressed to be the non-terminal $C$. If the nucleotide at position $j = 0$ in $S^i_o$ is to be retrieved, then that nucleotide is directly in the first position of the compressed sequence. If the nucleotide at position $j = 3$ in $S^i_o$ is to be retrieved, then that nucleotide is within the non-terminal $C$. Therefore, rule $C$ in the substitution rule dictionary $R$ needs to be examined. A mapping between each nucleotide in a sequence $S^i_o$ and where it maps to in the compressed sequence $S^i_c$ is necessary to retrieve nucleotides from
5.3. THE DISPLAY() QUERY

Figure 5.15: The mapping between sequence $S_o^1$ and the corresponding compressed sequence $S_c^1$. The substring highlighted in grey is the repeated substring replaced by non-terminal $C$. The set of substitution rules are shown on the right.

Algorithm 3 display_nucleotide($R, S, p$) retrieves a single nucleotide from a COMRAD-compressed sequence. $R$ is the set of substitution rules, $S$ is a COMRAD-compressed string and $p$ is the position of the nucleotide in the uncompressed string.

1: $p' \leftarrow \text{get\_compressed\_position}(S, p)$  \{pos in $S$ where nucl at $p$ occurs\}
2: if $S[p']$ is a nucleotide then
3:   return $S[p']$
4: else  \{nucleotide at $p$ is in a non-terminal\}
5:   $nt \leftarrow S[p']$
6:   $pnt \leftarrow p - \text{nonterm\_startpos}(S, nt)$  \{pos of $p$ in the substring of $nt$\}
7:   return display_nucleotide($R, R[nt], pnt$)  \{recursively retrieve the nucleotide\}
8: end if

the compressed sequence. Suppose there is a function get\_compressed\_position($i, j$) that implements this mapping. For a sequence $S_c^1$ and a position $j$, it returns the position in the compressed sequence $S_c^1$ that contains the nucleotide at $S_o^1[j]$.

Given this mapping, it is trivial to retrieve a nucleotide from the compressed sequence if it does not belong to a non-terminal. However, if the nucleotide belongs to a non-terminal, then the correct offset into the substring represented by the non-terminal needs to be determined to access it. For our example in Figure 5.15, the nucleotide at $S_o^1[3]$ occurs at position 2 of the substring represented by $C$. If the position in $S_o^1$ where the non-terminal $C$ starts is known, then the offset into the substring represented by $C$ can be calculated. In our example, $C$ starts at position 1 of $S_o^1$. Then the offset $k$ into the substring represented by $C$ is $k = 3 - 1 = 2$. Once this offset is known, the nucleotide can be retrieved by decompressing $C$ till the symbol at position 2 is retrieved. This may require other non-terminals to be decompressed recursively. The $C$ non-terminal represents the substring AgtB. Since non-terminal $A$ starts at position 0 in $C$ and it represents a substring of length 4, then, non-terminal $A$ contains the nucleotide of interest at position 2. Since, $A$ is a base level non-terminal, the nucleotide at position 2 can be retrieved directly from the substitution rule dictionary entry for $A$. The nucleotide $c$ is retrieved. In general, given a COMRAD-compressed string $S$, the nucleotide at a position $p$ can be obtained recursively as shown in Algorithm 3. The get\_compressed\_position() function is generalised to provide a mapping for any COMRAD-compressed string.

For a substring $S_o^1[s...e]$, it is possible to use Algorithm 3 to retrieve nucleotides
Algorithm 4 \( \text{display}(i, s, e) \) takes sequence \( i \), a start position \( s \) and an end position \( e \), and returns substring \( S_c^i[s \ldots e] \). Assume that the compressed sequences \( S_c \) and the substitution rule dictionary \( R \) are available to the function.

1: \( s' \leftarrow \text{get \_compressed \_position}(i, s) \) \{start pos of substring in compressed seq\}
2: \( \text{syms} \leftarrow 0 \) \{initialise the number of nucleotides retrieved\}
3: \( \ell \leftarrow s - e \) \{length of the substring to be retrieved\}
4: while \( \text{syms} < \ell \) do
5:   if \( S_i^c[s'] \) is a nucleotide then
6:     Retrieve as many nucleotides as possible from \( S_i^c \)
7:     Update \( s' \) and \( \text{syms} \)
8:   if \( \text{syms} == \ell \) then \{retrieved all nucleotides\}
9:     break
10: end if
11: end while
12: \( nt \leftarrow S_i^c[s'] \) \{nucleotides are in non-terminal\}
13: if \( \text{syms} == 0 \) then \{first nucleotide so may need to retrieve from an offset\}
14:   \( \text{pnt} \leftarrow s - \text{nonterm \_startpos}(S_i^c, nt) \)
15:   Retrieve \( \min(|nt| - \text{pnt}, \ell) \) nucleotides from \( R[nt] \) position \( \text{pnt} \)
16:   \( \text{syms} \leftarrow \text{syms} + \min(|nt| - \text{pnt}, \ell) \)
17: else
18:   Retrieve \( \min(|nt|, \ell - \text{syms}) \) nucleotides from rule \( R[nt] \)
19:   \( \text{syms} \leftarrow \text{syms} + \min(|nt|, \ell - \text{syms}) \)
20: end if
21: \( s' \leftarrow s' + 1 \)
22: end while

individually at positions \( s, s+1, s+2, \ldots, e-1 \). However, there is a simpler algorithm using the property that the nucleotides of the substring will be consecutive in the compressed string. Once the position in the compressed sequence where nucleotide \( S_i^c[s] \) occurs is determined, the remainder of the substring can be accessed directly. For our example sequence in Figure 5.15, to access \( S_i^c[3 \ldots 6] \), we use the same process as before to determine that the nucleotide at \( S_i^c[3] \) is at position 2 of the non-terminal \( A \). Since non-terminal \( A \) represents a substring of length 4, another nucleotide \( c \) at position 3 of \( A \) can be retrieved. This gives the substring \( S_i^c[3 \ldots 5] \). The last nucleotide cannot be accessed from \( A \), but recall that non-terminal \( A \) was accessed through non-terminal \( C \). Therefore, to obtain the last nucleotide of the substring, return to non-terminal \( C \), and retrieve the nucleotide occurring after the \( A \) non-terminal. This nucleotide is \( g \) and this completes the query to retrieve the substring \( ccg \). Algorithm 4 shows the pseudo-code for the \( \text{display}(i,s,e) \) algorithm.

We are yet to determine how to map the positions in the original sequences to positions in the compressed sequences (the \( \text{get \_compressed \_position}() \) function). The starting position in the original sequence for a given non-terminal also needs to be determined (the \( \text{nonterm \_startpos}() \) function). Compressed bit vectors can be used to efficiently implement both these queries. For each sequence, we construct two bit vectors. The first bit vector \( B^i_0 \) with length \( n^i_0 = |S^i_0| \) consists of a 1 bit
Figure 5.16: The bit vectors $B^1_o$ and $B^1_c$ for the original and compressed sequences $S^1_o$ and $S^1_c$ that indicate the positions where the non-terminals start in the original and compressed sequences, respectively. The lengths of the substrings represented by each non-terminal is also displayed as $NTlen$.

at every position where a non-terminal begins in the original sequence. The second bit vector $B^i_c$ with length $n^i_c = |S^i_c|$ consists of a 1 bit at every position where there is a non-terminal in the compressed sequence. In other words, $B^i_o$ indicates where the non-terminals are with respect to the uncompressed sequence, and $B^i_c$ indicates where the non-terminals are in the compressed sequence. We also need to store the lengths of the substrings represented by the non-terminals. We identify the lengths using the variable $NTlen$. The two bit vectors for the sequence in Figure 5.15 and the lengths of the non-terminals are displayed in Figure 5.16.

Using the two bit vectors, the mapping between the positions in the original sequences and the compressed sequences can be implemented using $rank_1()$ and $select_1()$ queries described in Section 3.1.4, on $B^i_o$ and $B^i_c$. Given a position $s$ in sequence $S^i_o$, a $rank_1(B^i_o, s)$ query can be used to determine the number of non-terminals $n$ that have occurred before or at position $s$, and then a $select_1(B^i_o, n)$ query can be used to determine the position of the $n$th non-terminal. Let the position of the $n$th non-terminal in the uncompressed sequence be $ntp$. Note this is the $nonterm\_startpos()$ function. To access the $n$th non-terminal, a $select_1(B^i_c, n)$ query can be used to determine the position $ntp'$ in the compressed sequence where the non-terminal occurs, and the non-terminal can be accessed by $S^i_c[ntp']$. We do not yet know whether the substring covered by the $n$th non-terminal includes position $s$. Let $|nt| = NTlen[nt]$. The non-terminal represents the substring $S^i_o[ntp \ldots ntp + |nt|]$. If $s < ntp + |nt|$, then the position $s$ is within the non-terminal so position $s$ in the uncompressed sequence maps to position $ntp'$ in the compressed sequence. Otherwise, if $s \geq ntp + |nt|$, then the position $s$ in the uncompressed sequence occurs after the non-terminal $nt$ so the position $s$ maps to position $ntp' + s - (ntp + |nt|)$ in the compressed sequence. Algorithm 5 presents the pseudo-code for the mapping.

In summary, the random access query $display(i, s, e)$ for COMRAD-compressed sequences can be implemented by first determining the position in the compressed sequence $S^i_c$ that maps to position $s$, then starting at that position, retrieving the necessary nucleotides from the terminals and non-terminals in $S^i_c$. Next, we discuss the time complexity of the query and the space complexity of the index.
Algorithm 5 get_compressed_position\((i, s)\) retrieves the position in the compressed sequence \(S'_c\) that contains the position \(s\) in the uncompressed sequence \(S'_o\).

1: \(n \leftarrow \text{rank}_1(B'_o, s)\) \{number of non-terminals in \(B'_o[0\ldots s + 1]\)\}
2: if \(n == -1\) then \{no non-terminals occurred before or at \(s\)\}
3: \(\text{return } s\)
4: end if
5: \(ntp \leftarrow \text{select}_1(B'_o, n)\) \{position of the \(n\)th non-terminal in \(S'_o\)\}
6: \(ntp' \leftarrow \text{select}_1(B'_c, n)\) \{position of the \(n\)th non-terminal in \(S'_c\)\}
7: \(nt \leftarrow S'_c[ntp]\) \{retrieve the non-terminal\}
8: \(|nt| \leftarrow \text{NTlen}[nt]\) \{the length of the substring represented by \(nt\)\}
9: if \(s < ntp + |nt|\) then \{position \(s\) occurs within non-terminal \(nt\)\}
10: \(\text{return } ntp'\)
11: else \{position \(s\) occurs after non-terminal \(nt\)\}
12: \(\text{return } ntp' + s - (ntp + |nt|)\)
13: end if

5.3.2 Time and space complexity

We now analyse the algorithmic complexity of the display\((i, s, e)\) query for COMRAD. Algorithm 4 shows that one rank and several select queries are required to implement the display query. The time and space complexity of the rank and select queries depend on the compressed bit vector implementation. We chose the sdarray implementation by Okanohara and Sadakane [2007] for the \(B'_o\) and \(B'_c\) bit vectors. Recall from Section 3.1.4 that the sdarray implementation requires \(O(1)\) time for a select\(_1\) query, and \(O(\log \frac{n}{n'})\) time for a rank\(_1\) query, where \(n\) is the number of bits, and \(m\) is the number of 1 bits in the bit vector. Therefore, each \(B'_o\) and \(B'_c\) bit vector has a time complexity of \(O\left(\log \frac{n}{n'}t'\right)\) and \(O\left(\log \frac{n}{n'}t\right)\), respectively, for a rank\(_1\) query, where \(t'\) is the number of non-terminals in the compressed sequence \(S'_c\).

The first step of Algorithm 4 uses the get_compressed_position() function to obtain the starting position to decode from, as described in Algorithm 5. The algorithm has a rank\(_1\) query on bit vector \(B'_o\) (line 1), which requires \(O\left(\log \frac{n}{n'}t\right)\) time, and a select\(_1\) query each on \(B'_o\) and \(B'_c\) bit vectors (lines 5–6), both of which require constant time. The access to sequence \(S'_c\) (line 7) and NTlen array are also constant time operations (assuming the non-terminal lengths are pre-calculated). Therefore, determining the starting position to decode from requires \(O\left(\log \frac{n}{n'}t\right)\) time.

The algorithm needs to retrieve nucleotides directly from the compressed sequence, and from non-terminals. An uncompressed nucleotide can be retrieved directly from a compressed sequence \(S'_c\) in constant time. To retrieve a nucleotide from a non-terminal, it may be necessary to access at most \(d\) non-terminals, where \(d\) is the maximum recursive depth of the substitution rules. In the worst case, every nucleotide retrieved may require \(d\) non-terminal accesses. Therefore, accessing \(\ell'\) nucleotides will require \(O(d\ell')\) time. Notice that the maximum rule depth corresponds to the number of iterations of the algorithm \(K\), which is a small constant in practice.
5.3. THE DISPLAY() QUERY

Therefore, \( \ell' \) nucleotides can be retrieved from a non-terminal in \( O(\ell') \) time.

Overall, \( e - s \) nucleotides need to be extracted per query. Some of the sub-strings of nucleotides in the query are accessed directly from the compressed sequence \( S_c \), while the remaining substrings of the query are obtained by extracting \( t' \) non-terminals that span the query string, where \( t' \) is the average number of non-terminals accessed during a query, and it is a function of the query length. Therefore, the overall time complexity for accessing a query of length \( e - s \) is \( O(e - s + t' + \log \frac{n_i}{t'}) \) time. The LZ-End self-index [Kreft and Navarro, 2010, 2011] can extract a substring in \( O(e - s) \) time if \( e \) coincides with the end of a phrase. But in the worst case, the time complexity is \( O(e - s + h) \), where \( h \) is a measure of how nested is the LZ-End parsing. This is similar to our maximum recursive depth \( d \), but unlike \( d \), which is a constant in practice, \( h \) can be as large as the number of factors in the LZ77 parsing. However, in practice it is likely to be much smaller, so the equivalent query in LZ-End is likely to be faster than the COMRAD display() query. On the other hand, RLCSA has a complexity of \( O(t_{LF}(g + e - s)) \) for the display() query, where \( t_{LF} \) is the time for the LF-mapping, and \( g \) is the rate at which the suffix array is sampled [Mäkinen et al., 2009]. At most \( g \) extra symbol may need to be extracted before extracting the relevant substring. The \( t_{LF} \) factor will most likely make the query perform slower than the COMRAD query in practice.

To implement the display\((i, s, e)\) query, instead of producing the encoded compressed output, COMRAD must output the necessary data structures to implement the query. We identify this as the COMRAD index. The index size is larger than the standard compressed output, but this is the cost of providing the random access feature. Once the compressor has completed the substitutions, the compressed sequences, and the substitution rules need to be processed to construct the index.

The index needs to contain the compressed sequences \( S_c \) and the set of substitution rules \( R \). As stated in Section 5.2.4, it is difficult to know the space used by the compressed sequences or the set of substitution rules without knowing the repeat properties of the collection. Assuming there are \( r \) substitution rules in \( R \) and the number of symbols in the compressed sequences are \( N_c = \sum_{i=1}^{M} |S_c^i| \), then the space complexity of \( S_c \) and \( R \) is \( O(r + N_c) \).

The lengths of the substrings represented by each non-terminal \( NTlen \) are also part of the index. The lengths of non-terminals representing L-mers can be inferred. For pattern substrings, the lengths can be calculated recursively in a similar manner to a dynamic programming calculation. Therefore, not all the non-terminals need to be expanded fully to calculate the lengths. The \( NTlen \) array can be stored in \( O(r) \) space or to be more precise, \( N_P \log m \) bits, where \( N_P \) is the number of pattern non-terminals and \( m \) is the length of the longest expanded substring represented by a non-terminal.

The index also requires two bit vectors per sequence. Each \( B^i_o \) bit vector requires \( t^i \log \frac{n_i}{t^i} + O(t^i) \) bits, while each \( B^i_c \) bit vector requires \( t^i \log \frac{n_i}{t^i} + O(t^i) \) bits. We
can store all the $B_i$ bit vectors concatenated into a single bit vector $B_o$, using a constant amount of space to determine the sequence boundaries. Bit vector $B_o$ requires $T \log \frac{N_o}{T} + O(T)$ bits of space, where $T = \sum_{i=1}^{M} t_i$ is the total number of nucleotides in $S_c$. The $B'_i$ bit vectors can also be concatenated in the same manner to construct a single bit vector $B_c$, which requires $T \log \frac{N_c}{T} + O(T)$ bits of space.

Overall, the space usage of the COMRAD index that supports the $\text{display()}$ query requires approximately $T(\log \frac{N_o}{T} + \log \frac{N_c}{T}) + O(r + N_c + T)$ bits. Comparing the COMRAD space complexity to those of LZ-End and RLCSA is non-trivial, since the space complexity is dependent on the number of symbols in the compressed collection and this cannot be directly related to the number of factors the collection is parsed into by LZ-End or the size of the data structures required by RLCSA.

Next we experimentally verify the $\text{display()}$ query performance of COMRAD by comparing it to the $\text{display()}$ query of LZ-End and RLCSA self-indexes.

### 5.3.3 Experimental evaluation

The $\text{display()}$ query performance of COMRAD for various query lengths and collections is presented below. We discuss the experimental setup, followed by the results.

#### Test data and environment

We are interested in evaluating COMRAD’s ability to extract substrings from multi-sequence collections, and also in evaluating the effect the level of repetition of a collection has on query performance and memory use of the COMRAD index. Therefore, we chose the following three collections from dataset-rep; the highly repetitive $\text{Sson}$ collection, the moderately repetitive $\text{Spara}$ collection, and the less repetitive $\text{Athal}$ collection. We constructed a COMRAD index for each collection.

For each collection, we randomly generated 2000 queries each of lengths, 10, 100, 1,000, 10,000 and 100,000. The first 1000 queries were used to ensure that all the necessary information is cached before the remaining 1000 queries are timed. The time to retrieve the symbols for a query is measured using the $\text{gettimeofday()}$ function. The query time does not include the time taken to output the substring. We report the number of micro seconds required to extract a symbol, averaged over the latter 1000 queries for each query length. The queries for each length were invoked five times per collection, and the best result from the five runs is reported.

We chose the RLCSA and LZ-End self-indexes to compare the COMRAD random access query. At the time of conducting these experiments, these were the best publicly available self-index implementations, in terms of the query performance and index size. RLCSA [Mäkinen et al., 2009; Mäkinen et al., 2010] and LZ-End [Kreft and Navarro, 2010, 2011] were introduced in Sections 3.4.2 and 3.4.3, respectively. We built an index for each collection with the RLCSA and LZ-End implementations. Although RLCSA can build an index for multi-sequence collections, LZ-End can only
build an index for a single sequence file, therefore the index was constructed for the concatenated sequences, and the query files were adjusted accordingly. The RLCSA parameters used were a block size of 32, sample rate of 512, and support display and locate set to 1 and 0, respectively. The locate(query) was disabled to make the index comparable with that of COMRAD. However, there was no option in the LZ-End implementation to disable the locate(query), so the LZ-End index may be a larger index than the COMRAD and RLCSA indexes.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

Results

The random access query times for COMRAD, RLCSA and LZ-End are reported in Table 5.5. For the two most repetitive collections, Sson and Spara, LZ-End was the fastest to retrieve nucleotides for short queries, while COMRAD was the fastest for longer queries. LZ-End has to access an LZ77 factor per symbol retrieved, so as the number of symbols to be retrieved increases, the number of unlocalized memory accesses increase, making the retrieval slower. COMRAD also has to access more non-terminals for longer queries, but multiple symbols can be retrieved once the base-level non-terminals are reached. This locality of symbols makes COMRAD faster than LZ-End for longer queries. The good performance for long queries is particularly advantageous for retrieving whole sequences, which can be achieved with a display(i, 0, n_i^0) query, where n_i^0 is the length of sequence S_i^0. Even without the index representation, a sequence can be retrieved from the standard compressed output by decompressing the substitution rules and then the sequence of interest.

COMRAD was faster than LZ-End for the Athal collection, and this is expected for less repetitive collections, since more of the nucleotides are uncompressed and can be accessed directly from the compressed sequences. Also, the substitution rule depth is likely to be lower, so base-level non-terminals can be reached faster.

As predicted, the RLCSA display() query is slow compared to the others. During the LF-mapping, many unlocalized memory accesses are required to retrieve the nucleotides, and these accesses are further complicated by the need to decompress individual entries of the compressed suffix array, as discussed in Section 3.4.2

Table 5.6 reports the space occupied by the indexes. The sizes were obtained from the size() methods of each index. The COMRAD index is the most space consuming index, due to the overhead of storing the substitution rules.

5.3.4 Discussion

We showed that the compressed output of COMRAD can be modified to construct an index on which the display() query can be implemented. The results showed that the query performance of COMRAD is comparable, or better for larger query lengths
Table 5.5: $\text{display()}$ times in $\mu$sec/base for COMRAD, RLCSA and LZ-End for varying query lengths on the Sson, Spara and Athal collections.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Index</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sson</td>
<td>COMRAD</td>
<td>8.074</td>
<td>0.835</td>
<td>0.117</td>
<td>0.042</td>
<td>0.034</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>17.634</td>
<td>2.350</td>
<td>0.843</td>
<td>0.693</td>
<td>0.677</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.402</td>
<td>0.537</td>
<td>0.433</td>
<td>0.423</td>
<td>0.420</td>
</tr>
<tr>
<td>Spara</td>
<td>COMRAD</td>
<td>4.031</td>
<td>0.433</td>
<td>0.071</td>
<td>0.036</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>17.997</td>
<td>2.377</td>
<td>0.847</td>
<td>0.689</td>
<td>0.675</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.206</td>
<td>0.510</td>
<td>0.418</td>
<td>0.407</td>
<td>0.406</td>
</tr>
<tr>
<td>Athal</td>
<td>COMRAD</td>
<td>1.082</td>
<td>0.146</td>
<td>0.053</td>
<td>0.043</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>23.238</td>
<td>3.099</td>
<td>1.063</td>
<td>0.861</td>
<td>0.844</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.677</td>
<td>0.798</td>
<td>0.681</td>
<td>0.662</td>
<td>0.662</td>
</tr>
</tbody>
</table>

Table 5.6: The sizes in Mbytes of the indexes constructed for the collections, Sson, Spara and Athal, by COMRAD, RLCSA and LZ-End, to support the $\text{display()}$ query.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sson</th>
<th>Spara</th>
<th>Athal</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMRAD</td>
<td>62.71</td>
<td>72.82</td>
<td>271.41</td>
</tr>
<tr>
<td>RLCSA</td>
<td>54.69</td>
<td>47.35</td>
<td>175.92</td>
</tr>
<tr>
<td>LZ-End</td>
<td>54.71</td>
<td>55.58</td>
<td>262.29</td>
</tr>
</tbody>
</table>

compared to one of the best performing indexes, LZ-End. The main drawback of the COMRAD index is its size. The size of the indexes are much larger than the standard compressed output of COMRAD; 18.34, 17.47, 65.62 Mbytes for the compressed size, compared to 62.71, 72.82, 271.41 Mbytes for the index, for Sson, Spara and Athal, respectively. The difference between the compressed size and the index size is the cost of enabling the $\text{display}(i,s,e)$ query. The substitution rules occupy the most space of the index, hence more efficient ways to store the rules need to be explored.

We did not explore ways to enable the $\text{count}(p)$ and $\text{locate}(p)$ queries in a COMRAD index. The rule hierarchy of COMRAD is similar to that of a context-free grammar, so the approach taken by Claude and Navarro [2010] in the SLP self-index can be adopted to search for patterns in the substitution rules. However, this is made difficult by the fact that the substitution rules contain non-terminals and nucleotides. Also, unlike in the SLP self-index where the sequences only consist of non-terminals, the sequences in COMRAD contain both terminal and non-terminal symbols. So not only is it necessary to construct an index of the substitution rules to determine which rules contain a given pattern, the substrings of nucleotides in the sequences also need to be indexed to quickly determine the existence of a pattern. Enabling search queries on the COMRAD index will be pursued in future work.
5.4 Chapter summary

In this chapter, we presented the COMRAD algorithm, which is specifically tailored for large multi-sequence DNA compression. The algorithm is a disk-based, semi-static dictionary compressor that detects repeats occurring across multiple sequences. The aim was to determine whether global repeat detection is feasible for large collections, and the experimental results confirmed the feasibility of this approach for multi-gigabase collections, as long as the collections are repetitive. The main drawback of the algorithm is its significant memory use, which is typical for semi-static dictionary compressors conducting global repeat detection. The compression performance of the algorithm was reasonable, and further improvements can be made to optimise the encoding techniques used to represent the compressed output. The compression and decompression speeds were comparable to those of the existing compression algorithms. Suggestions for reducing the memory use of the algorithm, and to improve the compressed size by using alternate encoding techniques were also discussed.

The modifications to the compressed output of COMRAD required to enable access to substrings from compressed collections was also discussed. The size of the modified compressed output was 3–4 times larger than the standard compressed output for our test collections. But the ability to query the compressed collection is important, especially for large collections, where the overhead of decompressing the entire collection is high when retrieving short substrings. The COMRAD display($i, s, e$) query was fast compared to the equivalent query implementations in the RLCSA and LZ-End self-indexes, especially for longer queries.

COMRAD is able to compress any collection of repetitive sequences even if the sequences are not related in terms of the function they serve or the species they belong to. Certain sequencing projects output multiple sequences from the same species or sequences known to serve the same function. Such collections are highly similar, where each sequence contains a few variations with respect to another sequence in the collection. In the next chapter, we introduce an algorithm known as RLZ that can compress such collections more efficiently than COMRAD, by using the similarity between a sequence chosen as the reference and every other sequence in a collection.
CHAPTER 5. EFFICIENT GENOME STORAGE WITH COMRAD
Chapter 6

Efficient genome storage with RLZ*

Resequencing is an important application of high-throughput sequencing. Many genomes of individuals from the same species can be sequenced at high speed and analysed to determine the variations in the genomes. Not only complete genomes, but genes or other segments of DNA from individual genomes can be resequenced to analyse the variations in the segment of interest. This type of analysis has many uses for medical research and more generally, for gaining a better understanding of the genetic makeup of various species. In Section 2.2.3, we discussed some resequencing projects such as the 1000 Genomes project and the 1001 Genomes project. Individual genomes from the same species or sequences from the same region of multiple genomes tend to be highly similar as discussed in Section 2.1.4. The excellent compression results for the Scere, Spara, and Sson collections from dataset-rep in Chapter 4 confirmed that genomes from the same species are indeed very similar. In this chapter, we introduce an algorithm designed to efficiently store sets of highly similar sequences, such as the output sequences of resequencing projects.

Most compression algorithms discussed so far build a model of the repeats in the input to compress it. The model is either constructed adaptively by algorithms such as gzip, 7-Zip and Sequitur, or semi-statically by algorithms such as Re-pair and COMRAD. For both adaptive and semi-static algorithms, the best compression results were produced when the repeats occurring globally are included in the model. For large sequence collections, such as those produced by DNA sequencing projects, managing global repeats require a significant amount of memory, as observed in Chapters 4 and 5. For collections such as Scere consisting of multiple individual genomes from the same species, a single genome is likely to contain repeats observed in the remaining genomes. Therefore, it may be unnecessary to build a model of repeats on the complete collection, and instead, a single genome can be used as the model. The remaining genomes can then be compressed with respect to the chosen

*Earlier versions of research in this chapter were published in Kuruppu et al. [2010, 2011a].
genomic. We identify this method of compression as relative compression, and the genome or sequence selected as the model to be the reference string or the reference. For collections produced by resequencing projects, relative compression is likely to be faster than using algorithms like 7-Zip, Re-pair, Sequitur and COMRAD, since the cost of compression is reduced to detecting repeats in the chosen reference. However, it is reasonable to expect comparable compression performance given that the reference is likely to contain a majority of the repeats in the collection.

Relative compression for DNA sequences was first proposed by Grumbach and Tahi [1993]. They defined this concept as ‘vertical compression’ and presented results for compressing each sequence in a set of sequences from five bacterial species, with respect to every other sequence in the set [Grumbach and Tahi, 1994]. For sequence pairs that are closely related phylogenetically, the compression ratio was better than for distantly related sequence pairs. CDNA [Loewenstern and Yianilos, 1997] can also be used for relative compression even though it is not specifically a relative compressor. The model can be trained on the reference, and the remaining sequences can be compressed with respect to this model. GenCompress [Chen et al., 2000] and XMCompress [Cao et al., 2007] also implement a relative compression feature.

Another variation of the approach is to assume that each sequence in the collection is identical to the reference, except for a few variations, and store just the differences with respect to the reference. The differences are typically SNPs and indels. This was the motivation behind the compression algorithms by Christley et al. [2009] and Brandon et al. [2009]. In Section 3.3.3, we identified several problems with this approach, including the cost of determining where the variations are, and the assumptions about the types of variations occurring between genomes. Algorithms like CDNA, GenCompress and XMCompress do not make such assumptions and can compress any DNA sequence with respect to another sequence. The main drawback of these algorithms is the slow compression speed, as discussed in Chapter 4.

In this chapter, we introduce a fast relative compressor known as RLZ (Relative Lempel-Ziv), designed to compress multi-sequence collections with high similarity. The algorithm is relatively simple and produces good compression results for large collections. It has fast compression and decompression speeds and uses less memory than existing DNA compressors. Like, CDNA, GenCompress and XMCompress, RLZ does not make assumptions about the types of repeats or variations in the collection being compressed. The algorithm is presented in Section 6.1, including an experimental analysis showing its excellent compression performance.

Then in Section 6.2, we present some modifications to RLZ that significantly improve compression results. The standard RLZ parsing is a greedy approach, and we show that a simple non-greedy approach can achieve better compression, without significantly increasing the computational complexity. We also propose two modifications to the encoding mechanism that further improve compression results. One of the improvements exploits a property specific to genomic collections. The
6.1 Relative Lempel-Ziv (RLZ) compression

We now present the Relative Lempel-Ziv (RLZ) algorithm in detail. The algorithm takes a set of DNA sequences as input, one of which is the reference string. The algorithm then encodes each sequence with respect to the reference using an LZ77-type encoding. Recall from Section 3.2.2 that the LZ77 algorithm encodes a repeated substring as a pointer to an earlier occurrence of the substring. RLZ adopts this method by encoding substrings shared between a sequence and the reference as pointers to the occurrences in the reference. The modified LZ77 algorithm is presented in Section 6.1.2. First we define the type of input expected by the RLZ algorithm.

6.1.1 Inputs

RLZ expects two or more DNA sequences as input. Let the input set of sequences be $T$, where $T$ contains $r$ sequences. Each sequence $T_i \in T$ has $n_i$ bases and the complete collection contains $N = \sum_{i=1}^{r} n_i$ bases. The symbols in each sequence must belong to the extended DNA alphabet $\Sigma$. For simplicity, we assume that the first sequence $T_1$ in the dataset $T$ is the reference string, and sequence $T_i$, where $2 \leq i \leq r$, is compressed with respect to $T_1$. Figure 6.1 shows an example set of sequences for RLZ. The sequences are similar to the reference $T_1$ and the differences of each sequence with respect to $T_1$ are underlined.

RLZ makes no assumptions about the types of repeats shared between the sequences being compressed and the reference, nor where they occur. Also no assumptions are made about the types of variations in each sequence with respect to the reference. The only assumption made is that the reference and the remaining sequences share the same alphabet. In terms of compressibility of the collection, the higher the level of similarity, the better the compression. Therefore, the best results are produced for collections containing sequences from the same species or serving the same biological functions, such as the output of resequencing projects.
6.1.2 Compression algorithm

The RLZ algorithm compresses each sequence in the input collection by detecting repeats in the sequence with respect to the reference using an LZ77 parsing. In Section 3.2.2, we introduced the standard LZ77 parsing, where repeats are detected with respect to the input seen so far. RLZ uses a modified version of the LZ77 parsing to detect repeats in each sequence with respect to the reference. The algorithm is presented in two steps. First, we discuss the modified LZ77 parsing algorithm, and then present the RLZ algorithm using this modified parser. Then we present the techniques used to compress the output of the parsing to produce the final compressed output of RLZ. We then discuss the time and space complexity of RLZ, followed by a discussion of the impact of mutations on compression.

Relative Lempel-Ziv factorisation

Relative Lempel-Ziv factorisation was first introduced by Ziv and Merhav [1993] for calculating the relative entropy between two strings. The relative Lempel-Ziv parsing is defined as follows. Let $X$ and $Y$ be two strings. The symbols in $Y$ are from an alphabet $\alpha$. The relative Lempel-Ziv factorisation of the string $X$ with respect to string $Y$, denoted as $LZ(X|Y)$, is a concatenation of a set of strings $w_0w_1w_2\ldots w_z$, where each $w_i$ belongs to one of the following two categories:

1. $w_i$ is a symbol not in $\alpha$.
2. $w_i$ is the longest prefix of the $|w_0\ldots w_{i-1}|$th suffix of $X$ that also occurs in $Y$.

Two example strings $X$ and $Y$ are shown in Figure 6.2. We identify the $w_i$ substrings as factors of $X$ with respect to $Y$. It is inefficient to express the factors as strings, so each factor is represented as a position and length pair, denoted by $(p, \ell)$. The position component $p_i$ of a factor $w_i$ is the position in $Y$ at which the substring $w_i$ was found. There may be multiple occurrences of $w_i$ in $Y$ and any one of the occurrences can be chosen for the factor. In practice, we choose the left-most occurrence in the reference. The length component $\ell_i$ is the length of the substring $w_i$. If a factor belongs to the first category, then it is represented as $(w_i, 0)$, where the length component of 0 indicates that it is not a substring that occurs in $Y$. The last line of Figure 6.2 shows the factorisation in terms of position and length pairs.

For convenience, we assume that no factors belong to the first category, where a symbol is encountered that is not in the alphabet $\alpha$. In other words, factors such as $(c, 0)$ from our example are assumed not to occur. If there are any symbols in $X$ that does not occur in $Y$, those symbols can be attached to the end of $Y$.

We now describe an algorithm that can be used to conduct a relative Lempel-Ziv parsing of a string $X$ with respect to $Y$. A suffix array of the reference string $Y$ is used to search for matching substrings between $X$ and $Y$. Suffix arrays were
Figure 6.2: The relative Lempel-Ziv factorisation of $X$ with respect to $Y$ in the form of strings, $w_0w_1w_2$, and in terms of position and length pairs.

described in Section 3.1.3. Recall that in a suffix array, suffixes with the same prefix are adjacent. Therefore, given a substring $p$, two binary searches on the suffix array can be used to determine the range in the suffix array where $p$ occurs. This is the property of the suffix array we use to conduct the relative Lempel-Ziv factorisation.

Let $SA_Y$ be the suffix array of string $Y$. Starting at position $j = 0$ of string $X$, we attempt to find the longest prefix of $X[j...|X|]$ that also occurs in $Y$. First we use two binary searches to find the range in $SA_Y$ that contains all occurrences of substring $X[j...j+1]$. Let this range be $SA_Y[lb_1...rb_1]$. Then we attempt to find the substring $X[j...j+2]$ in $Y$. If $X[j...j+2]$ exists in $SA_Y$, then it must be within the range $SA_Y[lb_1...rb_1]$, therefore, we continue to binary search using $lb_1$ and $rb_1$ as the starting range. Let $SA_Y[lb_2...rb_2]$ contain the substring $X[j...j+2]$. The process of binary searching is continued until a prefix of $X[j...|X|]$ is encountered that is not in $Y$. Let $k+1$ be the length of the smallest prefix of $X[j...|X|]$ that does not occur in $Y$. Then $X[j...j+k]$ is the longest prefix that occurs in $Y$. A position $p_0$ from the range $SA_Y[lb_k...rb_k]$ is chosen, and the factor is represented as $(p_0, \ell_0)$, where $\ell_0 = k$. Next, the algorithm sets $j = j + k$ and attempts to factorise the remainder of the string $X[j...|X|]$. The algorithm continues until the entire string is factorised. Figure 6.3 shows an example of the process used to find the first factor of sequence $T_2$ with respect to $T_1$. Figure 6.4 shows the factors produced for all the input sequences in Figure 6.1.

The algorithm for relative Lempel-Ziv factorisation of string $X$ with respect to string $Y$ is shown in Algorithm 6. The $j$ and $\ell$ variables are initialised to retrieve the first prefix of the string (line 1). Initially, the range in the suffix array to start searching for the prefix is set to be the entire suffix array (line 2). The first step to find a factor is to retrieve the left and right boundaries for the current prefix. This is done in Step 1 (lines 5–6) using the binary_search_left() and binary_search_right() functions, where existing boundaries are used to improve the efficiency of the search. Once the boundaries are found, it is possible to determine if the prefix can be extended further. If there are suffixes in $SA_Y$ with prefix $X[j...j+\ell]$ (Step 2 lines 7–9), then the prefix is extended by one symbol and the algorithm continues to find a longer prefix. If the longest prefix has already been found (when the prefix that is longer by one symbol is not found in the suffix array), the factor representing this
$T_1: \text{acacgacttttacgtatctt}$

$T_2: \text{acaccacttttacgtattt}$

<table>
<thead>
<tr>
<th></th>
<th>acacc</th>
<th>acacc</th>
<th>acacc</th>
<th>acacc</th>
<th>acacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>$</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>0</td>
<td>acacgacttttacgtatctt$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>acgacttttacgtatctt$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>acgtatctt$</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>acttttaacgtatctt$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>atctt$</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>cagacttttacgtatctt$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>cgacttttacgtatctt$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>cgtatctt$</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>ctt$</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>ctttaacgtatctt$</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>gacttttacgtatctt$</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>gtatctt$</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>t$</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>tacgtatctt$</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>tatctt$</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>tctt$</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>18</td>
<td>tt$</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>ttcgtatctt$</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>tttactttacttt$</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>ttttaacgtatctt$</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Factor for acac : $(0,4)$

$$LZ(T_2 | T_1) : (0,4) (1,3) (7,8) (10,3) (18,2)$$

Figure 6.3: An example factorisation for compressing sequence $T_2$ with respect to $T_1$. The two left-most columns represent the suffix array $SA_{T_1}$. The third column shows the range in $SA_{T_1}$ containing the substring $a$, the first prefix of $T_2$. The fourth column shows the range in $SA_{T_1}$ containing the substring $ac$. The search in the suffix array continues until the longest matching prefix $acac$ is found and the corresponding factor $(0,4)$ is output. A similar process continues until the whole sequence is factorised into the set of factors specified in the last line.

matching prefix is encoded (Step 3 lines 10–13). The $j$ and $\ell$ variables are updated to detect repeats for substrings occurring after the last encoded substring (line 12). The algorithm continues to find repeats and encode them as factors until the entire input string is factorised (lines 3–15).

To factorise the sequences in our input DNA sequence collection $T$, the same algorithm can be applied. A suffix array $SA_{T_1}$ is built for the reference $T_1$. Then each sequence $T_i$, where $2 \leq i \leq r$, is encoded using a call to Algorithm 6. In our implementation, we assume that the sequences in $T$ are derived from the alphabet.
6.1. RELATIVE LEMPEL-ZIV (RLZ) COMPRESSION

\[ T_1 : \text{acacgacttttacgtatctt} \]
\[ T_2 : \text{acaccacttttacgttactt} \]
\[ LZ(T_2|T_1) : (0,4) \, (1,3) \, (7,8) \, (10,3) \, (18,2) \]
\[ T_3 : \text{acacgacttttcggcgacgtatctt} \]
\[ LZ(T_3|T_1) : (0,11) \, (3,2) \, (4,1) \, (3,4) \, (13,7) \]
\[ T_4 : \text{acacgacttttacgtt} \]
\[ LZ(T_4|T_1) : (0,14) \, (18,2) \]
\[ T_5 : \text{aacgtattttcacgactttt} \]
\[ LZ(T_5|T_1) : (0,1) \, (11,6) \, (8,3) \, (1,7) \, (17,3) \]

Figure 6.4: The factorised output for the input sequences in Figure 6.1. The factors that represent mutations are in bold font.

\[ \sigma' = \sigma \cup \{n\}. \] We do not permit the remaining nucleotides from the extended DNA alphabet to be present in the input collection, since the overhead of storing these extra nucleotides are not justified given their rarity in practice. If nucleotides other than those from the alphabet \( \sigma' \) occur in \( T \), they are expected to be replaced with the nucleotide \( n \). Since many \( n \) nucleotides tend to occur consecutively, compression gains can be made by run-length encoding consecutive \( n \) nucleotides. Therefore, a series of \( n \) nucleotides are encoded as a factor \((n_1, \ell_N)\), where \( \ell_N \) is the length of the run of \( n \), and the position of the factor is the length of the reference, which will not be the position of any factor, so runs of \( n \)s can be unambiguously decoded. Next, we discuss the encoding techniques used to compress the factors.

**Factor encoding**

To compress a collection \( T \) of \( r \) sequences, first Algorithm 6 is used to factorise each sequence \( T_i \) with respect to \( T_1 \). To store the compressed sequences in \( T \), the reference and the factors for the remaining sequences are encoded. Since the reference is a string of symbols, it can be encoded using any general-purpose or DNA compression algorithm. We chose 7-Zip, since it achieves good compression for DNA sequences and unlike DNA compression algorithms, it is widely available.

The position and length components of each factor also need to be encoded. Let \((p, \ell)\) be a factor to be encoded. Since the position that a substring in sequence \( T_i \) can match to in \( T_1 \) is not restricted, the \( p \) value can range between 0 and \( n_1 \) (since \( n \) nucleotides are matched to position \( n_1 \)). Therefore, the position component is encoded using \( \lceil \log(n_1) \rceil \) bits. In Section 6.2, we show that the position of a factor can be predicted from the position of an earlier factor, which allows a factor to be encoded more efficiently. The length component of a factor, \( \ell \) can range in value from
Algorithm 6 $\text{relative\_LZ\_factorise}(X, Y, SA_Y)$ outputs a series of factors after an LZ parsing of string $X$ with respect to string $Y$. The inputs are, the string to be factorised $X$, the reference string $Y$ and the suffix array of the reference $SA_Y$.

1: $j \leftarrow 0$, $\ell \leftarrow 1$
2: $plb \leftarrow 0$, $prb \leftarrow |Y + 1|$
3: while $j < |X|$ and $j + \ell \leq |X|$ do
4: {Step 1: get left and right boundaries of $SA_Y$ with prefix $X[j \ldots j + \ell]$}
5: $lb \leftarrow \text{binary\_search\_left}(X[j \ldots j + \ell], Y, SA_Y, plb, prb)$
6: $rb \leftarrow \text{binary\_search\_right}(X[j \ldots j + \ell], Y, SA_Y, plb, prb)$
7: if $rb - lb > 0$ then {Step 2: suffixes exist in $SA_Y$ with prefix}
8: $\ell \leftarrow \ell + 1$
9: $plb \leftarrow lb$, $prb \leftarrow rb$
10: else if $rb - lb \leq 0$ or $j + \ell == |X|$ then {Step 3: found longest prefix}
11: $\text{encode}(SA_Y[plb], \ell)$
12: $j \leftarrow j + \ell$, $\ell \leftarrow 1$ {move to factorise unencoded part of string}
13: $plb \leftarrow 0$, $prb \leftarrow |Y + 1|$ {reset left and right search boundaries for $SA_Y$}
14: end if
15: end while

1 to $\max(n_1, n_i)$ (the sequence $T_i$ can be identical to $T_1$ or it can be a string of all $ns$). In practice, the length components tend to be much smaller than $\max(n_1, n_i)$, so it can be more efficiently encoded than using $\lceil \log(\max(n_1, n_i)) \rceil$ bits. In our implementation, we Golomb encode (Section 3.2.5) the length component. In our experiments we found that using a divisor $b = 64$ gives the best results. In this manner, the factors in each sequence $T_i$ are encoded and written to disk.

**Time and space complexity**

We now describe the time and space complexity of the RLZ compression algorithm. The algorithm consists of three parts; constructing a suffix array for the reference, generating factors for each sequence with respect to the reference, and encoding the reference and the factors for storage.

**Time complexity** As discussed in Section 3.1.3, a suffix array can be constructed in $O(n)$ time for a string of length $n$. Therefore, the suffix array for $T_1$ can be constructed in $O(n_1)$ time. To factorise a sequence $T_i$, two binary searches in the suffix array $SA_T_i$ are required for every position in the sequence. Each binary search requires $O(\log n_1)$ time, and a binary search on every position of the sequence is necessary, therefore requiring $O(n_i \log n_1)$ time per sequence $T_i$. The time complexity to factorise the entire collection of size $N - n_1 \approx N$, is $O(N \log n_1)$.

Each factor can be encoded in constant time, so if sequence $T_i$ can be represented in $z_i$ factors, then encoding the factors would require $O(z_i)$ time. If the entire collection is factorised into $Z = \sum_{i=2}^{r} z_i$ factors where $Z \ll N$, then the complexity of encoding the factors is $O(Z)$. The time complexity for compressing
the reference depends on the algorithm chosen. Generally, the complexity of compressing the reference is negligible compared to the complexity for factorising the remaining sequences. Therefore, the overall time complexity of the RLZ algorithm is $O(n_1) + O(N \log n_1) + O(Z) = O(N \log n_1)$.

**Space complexity**  The suffix array of $T_1$ is constructed using $n_1 \log n_1$ bits. During factorisation, the reference $T_1$, the suffix array $SA_{T_1}$, and the sequence being compressed need to be in memory. String $T_1$ requires $\lceil \log |\sigma'| \rceil n_1$ bits of space, where each nucleotide is encoded using $\lceil \log |\sigma'| \rceil \approx \log |\sigma'|$ bits. Suffix array $SA_{T_1}$ requires $n_1 \log n_1$ bits, where each item is encoded using $\log n_1$ bits. Each sequence $T_i$ also requires $n_i \log |\sigma'|$ bits of space. However, only one sequence is held in memory at a time, so the largest space occupation occurs when the longest sequence in the collection is held in memory. Let $T_{i'}$ be the longest sequence, where $2 \leq i' \leq r$. Then RLZ uses $n_1 (\log |\sigma'| + \log n_1) + n_{i'} \log |\sigma'|$ bits of memory during compression.

**Compressed storage space**  The compressed collection is stored in $n_1 H_k(T_1) + 2Z \log n_1$ bits, where the reference is compressed close to its $k$th order entropy, and the positions and lengths of factors are encoded using $\log n_1$ bits each. As discussed earlier, the factors will use less than $2Z \log n_1$ bits, since the length components are Golomb encoded to use less than $\log n_1$ bits each.

**Effects of mutations on factorisation**

The above analysis was conducted without making assumptions about the similarity between the sequences being compressed and the reference. RLZ aims to compress homologous sequences, where the sequences are derived from the same species or from sequences that serve the same biological function. Such sequences are identical except for the presence of mutations. We analyse how the presence of four types of mutations, single point mutations, insertions, deletions and rearrangements, affect factorisation. Recall from Section 2.1.2 that a single point mutation is a single nucleotide change, an insertion or a deletion is an insertion or deletion of one or more nucleotides, respectively, and a rearrangement swaps two substrings in one sequence with respect to another homologous sequence.

**Point mutations**  Sequence $T_2$ of Figure 6.4 shows an example of a single point mutation, where the nucleotide at position 4 is changed to $c$, with respect to $T_1$. If this mutation did not exist, then the substring `acacgacctttacgt` could be encoded as a single factor $(0,15)$. But as a consequence of this mutation, the substring is encoded using three factors, $(0,4)$ $(1,3)$ $(7,8)$. In general, the introduction of a mutation results in two factors being generated; one to cover the mutation and the other to cover the region beyond the mutation. For example, for the substring that is homologous to `acacgacctttacgt` in sequence $T_2$, factor $(1,3)$ covers the
mutation and factor \((7,8)\) covers the region beyond the mutation. Note that the factor encoding the mutation consists of 3 nucleotides. This is an artefact of the greedy factorisation, where the algorithm attempts to find the longest possible match in the reference. Figure 6.4 also shows that \(T_2\) has two other mutations that are adjacent to each other, the nucleotides \(ta\) at positions 15–16. Since two mutations are in close proximity, the algorithm found a substring in \(T_1\) that covered both the mutations, hence only two extra factors were introduced by these two mutations. Therefore, a single point mutation may not always generate two factors.

Lemma 1. If a string \(X\) contains \(s\) single point mutations with respect to a string \(Y\), and if \(X\) contains no other mutations, then \(LZ(X|Y)\) produces at most \(2s + 1\) factors.

Proof. If a mutation does not occur in the first position of \(X\), then the first factor represents the substring until the first mutation. For each mutation \(m_j\), at most two factors are introduced; one factor to cover the mutation \(m_j\) and another to cover the region between the current mutation \(m_j\) and the next mutation \(m_{j+1}\), or the end of the string. If mutations \(m_{j+1} \ldots m_{j+k}\) occur in close proximity to the current mutation \(m_j\), then the factor covering \(m_j\) may also cover the regions with the mutations \(m_{j+1} \ldots m_{j+k}\), generating \(k\) less factors. Also, if mutation \(m_j\) occurs at the last position of string \(X\), then only one factor is introduced. In the worst case, factorising a string containing \(s\) mutations will produce at most \(2s\) factors, and including the first factor, the total number of factors is at most \(2s + 1\).

Insertions. Figure 6.4 the factorisation for the insertion \(cggcg\) in sequence \(T_3\). An insertion consists of one or more nucleotides, and one or more factors are required to cover the inserted substring. For our example, the insertion requires only three factors \((3,2)\), \((4,1)\) and \((3,4)\) but in practice, at most six factors may be required to represent the insertion of length 6. Additionally, another factor is required to represent the region beyond the insertion, which is factor \((13,7)\) in our example.

Lemma 2. If a string \(X\) contains \(s\) insertions with respect to a string \(Y\), the sum of the insertion lengths is \(k\), and if \(X\) contains no other mutations, then \(LZ(X|Y)\) produces at most \(k + s + 1\) factors.

Proof. The insertions with a total length of \(k\) will require at most \(k\) factors to represent them, since it is plausible that each individual symbol in the insertions require a separate factor. If an insertion does not occur at the beginning of string \(X\), then a factor is needed to represent the substring until the first insertion. Each subsequent insertion requires a factor to represent the region from the end of the current insertion until the beginning of the next insertion, and if the last insertion occurs at the end of the string, then no factor is introduced. In the worst case, no insertions occur at the beginning or end of the string. So at most \(k + s + 1\) factors are required to represent a string containing \(s\) insertions of total length \(k\).
Deletions  Figure 6.4 shows a deletion of the substring tatc in sequence $T_4$ with respect to $T_1$. The deletion has the effect of introducing an extra factor to represent the substring from the deleted region to the end of the sequence, namely, factor $(18,2)$ in our example.

Lemma 3. If a string $X$ contains $s$ deletions with respect to a string $Y$, and if $X$ contains no other mutations, then $LZ(X|Y)$ produces at most $s + 1$ factors.

Proof. If a deletion does not occur at the beginning of string $X$, then a factor is required to cover the region from the start of the string until the first deletion. Each subsequent deletion requires a factor to represent the region from the end of the current deletion until the beginning of the next deletion, and if the last deletion occurs at the end of the string, then no factor is introduced. In the worst case, no deletions occur at the beginning or end of the string, so at most $s + 1$ factors are required to represent a string containing $s$ deletions.

Rearrangements  Figure 6.4 shows a rearrangement of the substrings acgtat and cacgact in $T_5$ with respect to $T_1$. As a result of the rearrangement, four extra factors are necessary; two factors to represent the two rearranged substrings, $(11,6)$ and $(1,7)$, a factor to represent the region between the rearranged strings, $(8,3)$, and a factor to represent the region after the second rearranged string, $(17,3)$.

Lemma 4. If a string $X$ contains $s$ rearrangements with respect to string $Y$, and if $X$ contains no other mutations, then $LZ(X|Y)$ produces at most $4s + 1$ factors.

Proof. If a rearrangement does not occur at the beginning of string $X$, then a factor is needed to represent the region from the start of $X$ until the first rearranged substring. The $s$ rearrangements create $2s$ substrings, which will be represented by $2s$ factors. Note that each rearranged substring will occur completely in $Y$, as it is a substring from $Y$ that is displaced to another position in $X$. Therefore each substring can be represented by a single factor. The $s$ rearrangements also require at most $2s$ factors to represent the regions between the rearranged substrings. Only $2s - 1$ factors are required if one of the rearrangements occur at the end of the string $X$. Therefore, in the worst case, $4s + 1$ factors are required to represent a string containing $s$ rearrangements.

We discussed the effects on relative Lempel-Ziv factorisation in the presence of various types of mutations that occur in DNA sequence collections containing homologous sequences. In summary, the presence of a single point mutation can introduce at most two extra factors, an insertion of length $k$ can introduce at most $k + 1$ factors, a deletion can introduce at most one extra factor, and a rearrangement of two substrings can introduce at most four extra factors. Clearly, if the rate of mutations in a sequence is high with respect to a reference the lower the compression, due to the large number of factors. Next we present the RLZ decompression algorithm.
Algorithm 7 \( RLZ_{\text{decompress}}(T_c, r) \) takes a set of RLZ-compressed sequences \( T_c \) and a number of sequences to be decompressed \( r \), then decompresses each sequence to produce the original collection.

1: \( T_1 \leftarrow \text{decompress reference from } T_c \) \{Step 1\}
2: \( i \leftarrow 2 \) \{next sequence to be decompressed\}
3: while \( i \leq r \) do \{Step 2\}
4: \( T_{c_i} \leftarrow \text{retrieve compressed sequence } T_i \)
5: \( j \leftarrow 0 \) \{start position for first decompressed factor\}
6: while more factors to be decompressed in \( T_{c_i} \) do
7: \( (p, \ell) \leftarrow \text{decode next factor from } T_{c_i} \)
8: \( T_i[j \ldots j + \ell] \leftarrow T_1[p \ldots p + \ell] \) \{retrieve substring for factor from reference\}
9: \( j \leftarrow j + \ell \)
10: end while
11: output decompressed sequence \( T_i \)
12: \( i \leftarrow i + 1 \)
13: end while

6.1.3 Decompression algorithm

The RLZ decompression algorithm is simple and consists of two steps as shown in Algorithm 7; first the reference is decompressed (line 1), then the factors for each sequence are decompressed using the reference (lines 3–13). The reference may have been stored either uncompressed or compressed using some compression algorithm. If it is compressed, then it needs to be decompressed and stored in a format that allows substrings to be retrieved in \( O(l) \) time, where \( l \) is the length of the substring being retrieved. Storing the reference using \( n_1 \left\lceil \log |\sigma'| \right\rceil \) bits per base is appropriate for both fast retrieval and lowering memory usage. Since all remaining sequences are stored as a series of encoded factors, to decompress each sequence, a factor at a time is decoded to obtain the position and length pair \((p, \ell)\). Each decoded factor is then decompressed to output the substring \( T_i[p \ldots p + \ell] \). The process continues until all factors are decoded to produce the original set of sequences.

The decompression algorithm is linear in the length of the collection \( N \). Since each nucleotide of the \( r - 1 \) sequences can be retrieved from the reference in constant time, each sequence can be retrieved in \( O(n_i) \) time, where sequence \( T_i \) has length \( n_i \). The complexity for decompressing the reference depends on the compressor used. Most general-purpose compressors allow decompression to occur in \( O(n_1) \) time for a string of length \( n_1 \). Therefore, the overall time complexity for decompression is \( O(N) \). During decompression, RLZ only stores the reference in memory. The remaining sequences are read from disk, one factor at a time, which requires \( 2 \log n_1 \) bits. Each decoded factor is written to disk immediately. Therefore, the memory usage during decompression is restricted to \( n_1 \left\lceil \log |\sigma'| \right\rceil + 2 \log n_1 \) bits.

The RLZ decompression algorithm is fast and memory-efficient, provided the reference is not too large. Given that collections are likely to be decompressed more frequently than it is compressed, having a fast decompression speed is advantageous.
RLZ can also decompress a single sequence from the collection without decompressing the other sequences apart from the reference. For large collections containing many sequences or large individual sequences, having the ability to retrieve individual sequences will be much more efficient. RLZ can also be extended to retrieve substrings from sequences without decompressing the entire collection. This random access feature is discussed in Chapter 7.

Next, we analyse the compression performance of RLZ. Then in Section 6.2, we present some optimisations that allow RLZ to compress some of these test collections to almost half the size achieved by the basic RLZ algorithm.

6.1.4 Experimental evaluation

We now analyse the practical compression performance of RLZ using some of the repetitive collections chosen from dataset-rep. The test data and environment used for the experiments are discussed in the next section, followed by the compression results. The compression performance of RLZ is also compared to some of the DNA compressors that implement relative compression, as well as some of the best performing general-purpose and DNA compressors from Chapter 4.

Test data and environment

RLZ is designed for compressing collections containing genomes of individuals from the same species, or sequences that serve the same biological function. In general, the collection must contain homologous sequences in order to achieve better compression. In dataset-rep from Chapter 4, several of the collections fit this category, namely Ecol, Athal, Scere, Spara, Sson and Hsap, so these collections were used for the experiments. The Ecol, Scere, Spara and Hsap collections contain assembled complete individual genomes from the specified species. The Athal and Sson collections contain assembled contigs of individual genomes. All but one of the six collections contain a fully assembled reference genome, and these reference genomes were used as the references for these collections. The E. coli species does not have a reference genome, so the genome for the K12 strain was used as the reference. In Chapter 8 we analyse the effects on compression when different genomes are used as the reference. Three of the collections in dataset-rep, Hemo, Mito and Infl are not suitable for RLZ compression so they were not added to the test dataset. Techniques for compressing these collections using RLZ are discussed in Chapter 8.

Table 6.1 summarises the properties of the test dataset.

The RLZ implementation was analysed in terms of the three performance criteria described earlier in Section 4.2.1. These are: the final compressed size of each collection measured in Mbytes, the compression and decompression speeds measured in seconds, and the approximate maximum memory used in Mbytes. The compressed size for each collection was measured as the sum of the RLZ-compressed
CHAPTER 6. EFFICIENT GENOME STORAGE WITH RLZ

<table>
<thead>
<tr>
<th>Coll.</th>
<th>Num. seqs.</th>
<th>Size (Mbases)</th>
<th>Reference</th>
<th>Genbank accession</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>33</td>
<td>164.90</td>
<td>K12 genome</td>
<td>U00096</td>
</tr>
<tr>
<td>Spara</td>
<td>36</td>
<td>429.27</td>
<td>S. paradoxus ref. genome</td>
<td>N/A</td>
</tr>
<tr>
<td>Scere</td>
<td>39</td>
<td>485.87</td>
<td>S. cerevisiae ref. genome</td>
<td>N/A</td>
</tr>
<tr>
<td>Athal</td>
<td>5</td>
<td>506.67</td>
<td>A. thaliana ref. genome</td>
<td>NC_003070–NC_003071, NC_003074–NC_003076,</td>
</tr>
<tr>
<td>Sson</td>
<td>231</td>
<td>966.46</td>
<td>S. sonnei ref. genome</td>
<td>NC_007384</td>
</tr>
<tr>
<td>Hsap</td>
<td>4</td>
<td>12066.06</td>
<td>H. sapiens ref. genome</td>
<td>NC_000001–NC_000024</td>
</tr>
</tbody>
</table>

Table 6.1: The collections used for the RLZ experiments. The columns are: the collection name, number of sequences in the collection, the number of bases in the collection (in Mbases), the sequence used as the reference, and its accession id, respectively. The S. cerevisiae and S. paradoxus reference genomes were assembled by the Saccharomyces Genome Resequencing Project.

The reference was compressed with the 7-Zip LZMA encoder, using the options -t7z -m0=lzma -mx=9 -mfb=64 -md=30 -ms=on. The suffix array for the reference was generated using the divsufsort implementation, which can construct a suffix array for a string of length $n$ in $O(n \log n)$ time using $O(1)$ extra working space above the $O(n)$ words of space required to store the string and the suffix array. The compression time was measured as the time taken to compress each sequence in a collection with respect to the reference using RLZ, the time taken to generate the suffix array for the reference, and the time taken to compress the reference using 7-Zip. The decompression time was measured as the time taken to decompress each sequence and to decompress the reference using the 7-Zip decompressor. The approximate maximum memory used was measured using the Valgrind massif tool during RLZ compression.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

Results

Table 6.2 contains the RLZ compression results for the test collections. The compressed sizes produced by RLZ are far better than that could be achieved by a naive encoding. Most collections were compressed to a level consistent with the repetition levels of the collections. To obtain the best compression results for Sson, the reverse complement of the reference had to be included in the reference, since some of the contigs in the collection were assembled for the reverse strand. The Athal collection of contigs also achieved good compression. The results also show that RLZ is fast to both compress and decompress. For the Hsap collection, the genomes were compressed a chromosome at a time to reduce the number of bits required to encode the positions, hence the relatively larger compression and decompression speed. The factors for each chromosomal sequence are only likely to originate from the equivalent
### 6.1. Relative Lempel-Ziv (RLZ) Compression

<table>
<thead>
<tr>
<th>Coll.</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decomp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>23.14</td>
<td>1.18</td>
<td>124</td>
<td>31.72</td>
<td>3</td>
</tr>
<tr>
<td>Spara</td>
<td>22.29</td>
<td>0.44</td>
<td>174</td>
<td>55.09</td>
<td>4</td>
</tr>
<tr>
<td>Scere</td>
<td>17.06</td>
<td>0.29</td>
<td>141</td>
<td>57.13</td>
<td>4</td>
</tr>
<tr>
<td>Athal</td>
<td>40.21</td>
<td>0.67</td>
<td>500</td>
<td>474.68</td>
<td>7</td>
</tr>
<tr>
<td>Sson</td>
<td>20.18</td>
<td>0.17</td>
<td>173</td>
<td>48.02</td>
<td>5</td>
</tr>
<tr>
<td>Hsap</td>
<td>716.05</td>
<td>0.50</td>
<td>10,084</td>
<td>1023.03</td>
<td>131</td>
</tr>
</tbody>
</table>

Table 6.2: RLZ compression results for the collections in dataset-rep. The columns are: the collection name, compressed size in Mbytes, average number of bits per base used, compression time in seconds, approximate maximum memory usage during compression in Mbytes, and decompression time in seconds, respectively.

Reference genome chromosome. It is possible for indels or rearrangements to occur across chromosomes. However, the compression gains of the approach we followed outweighed the ability to detect repeats across chromosomes.

Interestingly, RLZ did not outperform the best compressors from Chapter 4, namely 7-Zip, Sequitur, Re-pair, dna-x, and XMCompress, as shown in Figure 6.5. However, RLZ compressed the Hsap collection better than Sequitur or Re-pair, both of which compressed the collection in blocks due to the inability to keep the entire collection in memory. Although RLZ effectively compressed this collection in blocks, better compression was achieved. The better performance of the above-listed compressors is unsurprising given that a concatenated sequence of the collections were compressed to enable global repeat detection. RLZ on the other hand only detected repetitions between the sequence being compressed and the reference.

When comparing the compression speeds, RLZ is significantly faster than the tools mentioned above, especially for the larger collections. The compression speed of RLZ is much faster than that of the two DNA compressors, which could not even compress the Hsap collection due to the excessive compression times required for a large collection. Considering the relatively promising results produced by RLZ for the Hsap collection, and the excellent compression speed, RLZ looks to be ideal for large collection compression. The decompression speed of RLZ is also fast, and compares well to the general-purpose compressors. The memory use of RLZ is many orders of magnitude lower than for 7-Zip, Sequitur, Re-pair and XMCompress, while also being lower than that of dna-x. This is a significant feature of the algorithm making it scalable for compressing much larger collections.

When the RLZ results are compared to those of COMRAD from Chapter 5, RLZ produced slightly worse compression results than COMRAD for most collections, except for Athal and Hsap, as shown in Figure 6.5. Once again, this is expected since COMRAD detects global repetitions. The most likely reason why RLZ was able to compress the Athal and Hsap collections is because these collections contain only a
few sequences, and COMRAD is not able to select the best repeats to create substitution rules. As a result, the COMRAD codebook contains too many substitution rules. RLZ on the hand is much faster to compress than COMRAD, and much more memory-efficient. RLZ decompression is also slightly faster than COMRAD. These properties make the RLZ algorithm ideal for large collection compression, as long as the sequences are genomes from the same species, or other homologous sequences with high similarity.

We also experimented with the DNA compressors, GenCompress, XMCompress, and CDNA. Our attempts at using GenCompress as a relative compressor were unsuccessful due to a software problem while compressing the Ecol and Scere collections. For the Spara collection, after 30 minutes, only 18% of a single sequence was compressed, so it was infeasible to attempt to compress the remainder of the collection. For CDNA, we attempted to train a model using the reference, and compress every other sequence in the collection using this model. But even after one hour, CDNA had not completed compressing a single Ecol sequence, which is only around 5 Mbases in size. We therefore did not attempt to compress any larger collections with CDNA.

The XMCompress relative compression results are presented in Table 6.3. Relative compression does not produce results that are as promising as when the concatenated set of sequences are compressed, as in Table 4.8b. Intuitively, the repeats of the entire collection are not visible when just a single sequence is used as a reference. Although XMCompress produced better compression results than RLZ, it is a few hundred times slower to compress and decompress, and uses significantly more memory. Despite the excellent results, XMCompress cannot scale for large collection compression.

The compression results showed that RLZ can feasibly compress large collections
6.2 Optimised Relative Lempel-Ziv compression

While the experimental results in the previous section showed that RLZ can produce reasonable compression results, improvements are possible. We consider three optimisations to RLZ that produce better compression results. Two of the optimisations can be applied to any collection, while the third is only effective for collections with certain properties. The LZ factorisation algorithm adopted by the basic RLZ algorithm is a greedy method. Once a longest matching substring is found, it is encoded as a factor immediately, without considering whether longer factors could be found a few positions ahead. Therefore, in Section 6.2.1 we consider a non-greedy alternative of the LZ factorisation algorithm. The second optimisation we consider is the encoding of short factors. Currently, the algorithm uses the same encoding technique to compress all factors. However, some factors tend to be shorter than others, particularly those that encode mutations, and these factors can potentially be encoded more efficiently. Therefore, we consider the option of storing short factors as substrings rather than as position and length pairs in Section 6.2.2. The final optimisation is to encode factor positions more efficiently when compressing sequences that align to the reference. The alignment allows the position component of certain factors to be predictable, allowing RLZ to encode such positions using fewer bits. This optimisation will be discussed in Section 6.2.3. Although some of these optimisations will increase the compression time and memory usage, in some circumstances, the compressed sizes will almost be halved as will be seen in the
CHAPTER 6. EFFICIENT GENOME STORAGE WITH RLZ

\[
\begin{align*}
Y &: \text{gctcctatacgttatcctatg} \\
X &: \text{atcctatacgt} \\
LZ(X|Y) &: (13,7)(8,4) \\
LZ'(X|Y) &: (8,1)(2,10)
\end{align*}
\]

Figure 6.6: It may be possible to find longer factors by looking ahead, i.e. factor \((2,10)\) is detected by looking ahead by one position.

experimental results in Section 6.2.4. Each optimisation is discussed in detail below.

6.2.1 Non-greedy Lempel-Ziv factorisation

The basic RLZ algorithm presented earlier uses the relative Lempel-Ziv factorisation algorithm to compress each sequence \(T_i \in T_2 \ldots T_r\) with respect to the reference string \(T_1\). Recall that if \(m\) symbols in \(T_i\) are already factorised, then the next factor is obtained by finding the longest prefix of \(T_i[m \ldots |T_i|]\) that occurs in \(T_1\). Let the position and length of this factor be \(p_0\) and \(\ell_0\), respectively. The factor \((p_0, \ell_0)\) is encoded and the algorithm continues to factorise from position \(m + \ell_0\). This is a greedy algorithm, where a factor is encoded immediately after its found. However, it is possible that a longer factor could have been found beginning at position \(m + 1\) of \(T_1\). Let the position and length of this factor be \(p_1\) and \(\ell_1\), respectively, where \(\ell_1 > \ell_0\). Then encoding factors \((p_0, 1)\) and \((p_1, \ell_1)\) may result in a smaller compressed size than encoding factor \((p_0, \ell_0)\). This is a non-greedy algorithm and the term looking ahead will be used to refer to the process of checking for factors starting at many positions ahead of the current factorisation position.

An example of this behaviour is shown in Figure 6.6. The standard factorisation of string \(X\) with respect to string \(Y\) resulted in two relatively long factors \((13,7)\) and \((8,4)\). Instead of producing factor \((13,7)\), if the longest substring starting at the next position is considered, then a longer factor can be found. This factorisation also produces two factors, but one factor is short and the other is much longer. The compressed size is unaffected if the same encoding technique is used to compress all factors. However, if shorter factors can be encoded more efficiently, i.e. two bits to encode a single nucleotide factor, then the factors from \(LZ'(X|Y)\) are more compressible than the factors from \(LZ(X|Y)\).

In general, non-greedy Lempel-Ziv factorisation will not produce less factors or a smaller compressed sequence than when greedy factorisation is used. If a fixed number of bits is used to encode each factor, then using a greedy algorithm will result in just as good compression as using a non-greedy algorithm [Ferragina et al., 2009; Horspool, 1995]. However, if factors are encoded using a variable length encoder, then possible compression gains could be made. We intend to achieve compression
gains with the combination of non-greedy Lempel-Ziv factorisation and encoding shorter factors differently to longer factors. It should be noted that Ferragina et al. [2009] proposed an algorithm to produce a bit-optimal LZ77 parsing given a function $F$ to encode the position components of a factor, and a function $G$ to encode the lengths. Although our aim is to improve compression, we do not adopt this approach since optimising the factorisation according to some arbitrary encoding functions will not emphasise the relationship between factors and the biological properties of the sequences. Also the proposed method requires the entire collection to be in memory to determine the bit-optimal encoding, which is not feasible for larger collections.

Non-greedy factorisation is not just applicable to relative Lempel-Ziv factorisation, but is also applicable to general Lempel-Ziv factorisation. A well-known implementation that uses non-greedy factorisation is gzip, which looks ahead by one position. If the repeat starting at position $m + 1$ is longer than the repeat starting at position $m$, then the symbol at position $m$ is encoded as a single literal, and the repeat starting at position $m + 1$ is encoded as a pointer to the earlier occurrence of that repeat. Horsepool investigated this idea further by allowing an LZ compressor to look ahead by at most $h$ positions to find longer factors [Horspool, 1995]. Let $\ell_0$ be the length of the longest matching substring at position $m$. The longest matching substrings starting at each position from $m + 1$ to position $m + h$ are also found. Suppose that the longest matching substring starting at position $m + j$ has length $\ell_j > \ell_0$. Then the symbols at positions $m$ to $m + j - 1$ can be encoded as single literals or by some other means, while encoding the longest matching substring starting at position $m + j$ as a factor. Horsepool explored this approach on English text, where up to seven positions ahead were checked to find the best factor to encode. He found that in English texts, looking ahead by up to 5–6 positions resulted in improved compression but no improvements were made beyond that.

We implement two variants of non-greedy relative Lempel-Ziv factorisation for RLZ; one is equivalent to that of Horspool [1995], while the other permits looking ahead indefinitely until a locally maximal longest matching substring is found.

Non-greedy relative Lempel-Ziv factorisation algorithm

Here we describe the modifications made to the greedy relative Lempel-Ziv factorisation algorithm to allow for non-greedy factorisation. Let $X$ be a string that is to be factorised with respect to string $Y$. Let $SA_Y$ be the suffix array of $Y$ and let $h$ be the maximum look-ahead length. Assuming $j$ symbols of $X$ has been factorised already, first the longest prefix of $X[j \ldots |X|]$ also occurring in $Y$ is found. Let the factor corresponding to this longest matching substring be $(p_j, \ell_j)$. Next the longest prefix of $X[j + 1 \ldots |X|]$ also occurring in $Y$ is found. The factor corresponding to this substring is $(p_{j+1}, \ell_{j+1})$. This process continues until the factors starting at each of the positions in the range $j \ldots j + h$ are found. Let $\ell_k$ be the length of the
CHAPTER 6. EFFICIENT GENOME STORAGE WITH RLZ

\[ X : \text{cgacgtatctt} \]
\[ Y : \text{cgacttttacgtatctt} \]

<table>
<thead>
<tr>
<th>j</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(0,4)</td>
</tr>
<tr>
<td>1</td>
<td>(1,3)</td>
</tr>
<tr>
<td>2</td>
<td>(8,9)</td>
</tr>
<tr>
<td>3</td>
<td>(9,8)</td>
</tr>
<tr>
<td>4</td>
<td>(10,7)</td>
</tr>
<tr>
<td>5</td>
<td>(11,6)</td>
</tr>
</tbody>
</table>

\[ LZ'(X|Y) : (0,2) (8,9) \]

Figure 6.7: An example of the non-greedy relative Lempel-Ziv factorisation method to factorise string \( X \) with respect to string \( Y \). Let the look-ahead limit be \( h = 5 \). All factors starting from position 0 to position 5 are found. The longest matching substring occurs at position 2. Therefore, the substring until position 2 is represented by factor (0,2) and the remainder of string \( X \) is represented by factor (8,9).

In general, to compress a collection \( T \) with \( r \) sequences, each sequence \( T_i \), where \( 2 \leq i \leq r \), will be factorised with respect to \( T_1 \) using Algorithm 8. Then the factors could be encoded for storage using the encoding scheme described earlier; position components are encoded using \( \log n_1 \) bits each, and length components are Golomb coded. However, as discussed earlier, using the same encoding scheme for all factors will not necessarily result in compression gains when non-greedy factorisation is used. Shortly, we describe an encoding scheme for shorter factors to realise the compression gains of non-greedy factorisation.

One of the disadvantages of this method is the need to choose a look-ahead limit \( h \). If the limit chosen is too small, longer factors occurring beyond the look-ahead region may not be detected. Choosing too large a limit adds unnecessary complexity to the algorithm, since most of the time, more positions than necessary may need to be checked. In general, the algorithm may attempt to find a factor starting at a given position \( p \) at most \( h + 1 \) times when factorising begins from positions \( p-h, p-h+1, \ldots, p-1, p \). This can occur if position \( p \) is not included in any of the factors starting at positions between \( p-h \) and \( p-1 \). The complexity can be reduced by caching the factors found so far so that the algorithm only attempts
### Algorithm 8

*relative\_LZ\_factorise\_lookaheadH*(X, Y, SA_Y, h) takes as inputs, a string X to be factorised, reference string Y, suffix array of the reference SA_Y and look-ahead limit h, and outputs factors after a non-greedy relative Lempel-Ziv parsing of X with respect to Y.

1. \( j \leftarrow 0 \)
2. while \( j < |X| \) do \{X can still be factorised\}
3. \( m \leftarrow 0, \ p_m \leftarrow 0, \ \ell_m \leftarrow 0 \)
4. for \( k \leq h \) do \{Step 1: look ahead up to \( h+1 \) positions\}
5. \((p, \ell) \leftarrow \text{longest factor starting at } X[j+k] \)
6. if \( \ell > \ell_m \) then \{Keep track of the longest factor found so far\}
7. \( m \leftarrow k, \ p_m \leftarrow p, \ \ell_m \leftarrow \ell \)
8. end if
9. \( k \leftarrow k + 1 \)
10. end for
11. {Step 2: encode factors representing }\( X[j \ldots j + m + \ell_m] \)
12. encode factors for \( X[j \ldots j + m] \)
13. encode \((p_m, \ell_m)\)
14. \( j \leftarrow j + m + \ell_m \)
15. end while

To find the longest factor starting at a given position at most once. In the next section, we propose a modification to the non-greedy factorisation algorithm that detects local maximum factors without using a look-ahead limit.

#### Local maximum factor detection

We modify the non-greedy relative Lempel-Ziv factorisation presented in the previous section to detect local maximum factors in terms of the factor length. This method does not impose restrictions on the look-ahead limit. If a factor that is thought to be the local maximum factor (identified as the longest factor found so far) is found, its length can be used to adjust the look-ahead limit to ensure that it is indeed the local maximum factor in that region. In Figure 6.8, if factorisation of X with respect to Y begins at the first position of X, after looking ahead by two extra positions, the factor (8,9) is the longest factor found so far. Then the factors starting at eight more positions are checked to ensure that no other longer factor overlaps with this factor making it the local maximum. Therefore, the look-ahead limit for the first position of X is restricted to ten. After factorising this local maximum factor, factorisation continues from position 11. A longest factor (0,8) is found by looking-ahead by two positions. To check if this is the local maximum factor, the factors starting at a further seven positions are checked. In this case, the look-ahead limit at position 11 only needs to be nine. In this manner, the look-ahead limit can be adjusted based on the longest factor found so far.

The aim is to ensure that the local maximum factor in a region is represented...
Figure 6.8: String $X$ contains two long substrings that also occur in the reference string $Y$, which are underlined. These two substrings are local maxima in terms of the substring length and will be represented by a single factor each when $X$ is factorised with respect to $Y$.

using a single factor. We define the local maximum factor in a region as follows:

**Definition 5.** Let factor $(p', \ell')$ be a local maximum factor in the prefix $X[j \ldots |X|]$, where $j' \geq j$. Then no other factor beginning at positions in the ranges $[j, j')$ and $[j' + 1, j' + \ell')$ has a length $\geq \ell'$, making factor $(p', \ell')$ the local maximum factor in the region $X[j \ldots j' + \ell']$.

Only a small modification to the basic non-greedy factorisation algorithm is necessary to find the local maximum factor in a region. When factorising the prefix $X[j \ldots |X|]$ of string $X$, initially the factor $(p_j, \ell_j)$ starting at position $j$ of $X$ is found. To check if this is the local maximum factor, we set $h = \ell_j - 1$ and look-ahead by $h$ positions to find a longer factor. Suppose that at a position $j'$, where $j < j' \leq j + h$, another factor $(p_{j'}, \ell_{j'})$ is found where $\ell_{j'} > \ell_j$. Then factor $(p_{j'}, \ell_{j'})$ is the current longest factor, and the look-ahead limit is extended to $h = h + \ell_{j'} - 1$, to check if this is the local maximum factor. The algorithm continues until the actual local maximum factor is found. The factors starting at each position from $j$ until $j + h$ are cached in some structure $F$ so that the series of factors until the local maximum factor can be accessed for encoding purposes.

Suppose that when factorising the prefix $X[j \ldots |X|]$, the next local maximum factor $(p_m, \ell_m)$ is found at position $j_m \geq j$. Then the substring prior to where this factor starts, $X[j \ldots j_m]$, is encoded with one or more factors. The factors to cover the substring are retrieved from $F$ and the factors are chosen in a greedy manner. If the local maximum factor does not start at $j$, the short factor starting at $j$ is output, which covers the substring $X[j \ldots \min(j_m, j + \ell_j)]$. The $\min(j_m, j + \ell_j)$ calculation is necessary to ensure that a factor does not overlap with the local maximum factor. If the previous factor does not cover $X[j \ldots j_m]$ then the next factor that is output is the factor starting at $j + \ell_j$ retrieved from $F$. The process of factorising the substring $X[j \ldots j_m]$ continues until the entire substring is covered. Finally, the local maximum factor $(p_m, \ell_m)$ is output.

Algorithm 9 shows the modified Algorithm 8 that implements non-greedy factorisation without a specific look-ahead limit. The only difference is that when a new long factor is found, the look-ahead limit $h$ is updated (line 11).

The interesting consequence of this modification is that the RLZ algorithm now finds local maximum factors in terms of the factor lengths. The algorithm ensures that each substring corresponding to a local maximum factor is represented by a
Algorithm 9 \textit{relative\_LZ\_factorise\_local\_Max}(X, Y, SA_Y) takes as inputs, the string to be factorised \(X\), the reference string \(Y\) and the suffix array of the reference \(SA_Y\), and outputs factors of the non-greedy relative Lempel-Ziv parsing of \(X\) with respect to \(Y\).

1: \(j \leftarrow 0\)
2: \textbf{while} \(j < |X|\) \textbf{do} \{\(X\) can still be factorised\}
3: \(m \leftarrow 0\)
4: \((p_m, \ell_m) \leftarrow\) longest factor starting at \(X[j]\) \{\textbf{Step 1: initialise longest factor}\}
5: \(h \leftarrow \ell_m - 1\) \{initialise look-ahead limit\}
6: \(k \leftarrow 1\)
7: \textbf{while} \(k \leq h\) \{\textbf{Step 2: find factors until it is certain local max is found}\}
8: \((p, \ell) \leftarrow\) longest factor starting at \(X[j + k]\)
9: \textbf{if} \(\ell > \ell_m\) \textbf{then} \{keep track of the longest factor found so far\}
10: \(m \leftarrow k, p_m \leftarrow p, \ell_m \leftarrow \ell\)
11: \(h \leftarrow h + \ell_m - 1\) \{increase the look-ahead limit\}
12: \textbf{end if}\n13: \(k \leftarrow k + 1\)
14: \textbf{end while}\n15: \{\textbf{Step 3: encode factors representing }X[j \ldots j + m + \ell_m]\}\n16: \textbf{encode} factors that cover \(X[j \ldots j + m]\)
17: \textbf{encode} \((p_m, \ell_m)\)
18: \(j \leftarrow j + m + \ell_m\)
19: \textbf{end while}\n
single factor rather than being divided between two factors. In the context of DNA, the repeated substrings corresponding to the local maximum factors are likely to be the alignment regions between \(X\) and \(Y\). Therefore the substrings that are represented by the longest factors may have some biological significance.

In terms of the efficiency of the RLZ algorithm, the ability to encode the local maximum factors will not necessarily result in a better compressed size. As for the standard non-greedy factorisation, more factors maybe introduced by this modification than by the basic factorisation algorithm. If the encoding mechanism for the factors are not differentiated, then the compression results are likely to be worse. A method of efficiently encoding short factors will be discussed shortly.

For the standard non-greedy factorisation, in the worst case, the factors that start at all positions of the string may need to be found. On the other hand, for the non-greedy factorisation that finds the local maximum factors, the factors starting at all positions of the string need to be found in the best and worst cases of the algorithm. Next we relate this fact to the concept of \textit{matching-statistics}.

\textbf{Matching-statistics}

For non-greedy relative Lempel-Ziv factorisation, it may be necessary to calculate the factors starting at every position of the string being factorised. The \textit{matching-statistics} of a string \(X\) with respect to a string \(Y\) (denoted \(M(X|Y)\)) is a table of
position and length pairs \((p_j, \ell_j)\), where \(0 \leq j < |X|\), such that:

(a) \(X[j \ldots j + \ell_j]\) is the longest prefix of \(X[j \ldots |X|]\) that occurs in \(Y\), and

(b) \(X[j \ldots j + \ell_j] = Y[p_j \ldots p_j + \ell_j]\).

The concept of matching-statistics was first introduced by Chang and Lawler [1994]. For example, let \(X = \text{actaagactc}\) and \(Y = \text{actaacttc}\). Then \(M(X|Y)\) is:

<table>
<thead>
<tr>
<th>(j)</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
<th>(9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p_j, \ell_j)</td>
<td>(0, 5)</td>
<td>(1, 4)</td>
<td>(2, 3)</td>
<td>(3, 2)</td>
<td>(0, 1)</td>
<td>(0, 0)</td>
<td>(0, 3)</td>
<td>(1, 2)</td>
<td>(7, 2)</td>
<td>(1, 1)</td>
</tr>
</tbody>
</table>

The second column of the table denotes that the longest matching prefix of \(X[0 \ldots |X|]\) is at \(Y[0 \ldots 5]\). Similarly, the third column denotes that the longest matching prefix of \(X[1 \ldots |X|]\) is at \(Y[1 \ldots 4]\), and so on. A special case occurs at position \(j = 5\) in \(X\), where the symbol \(g\) does not occur in \(Y\). This event is indicated by the special factor \((0,0)\). Also, if more than one substring in \(Y\) matches a longest prefix of \(X\), then any one of the occurrences in \(Y\) can be chosen.

It is clear that both the greedy and non-greedy versions of the relative Lempel-Ziv factorisation of a string \(X\) with respect to a string \(Y\) outputs a subsequence of the factors in \(M(X|Y)\). Prior to Kuruppu et al. [2011a], to our knowledge, no previous link has been made between a Lempel-Ziv parsing of a string with respect to another string, and its matching statistics. Matching-statistics are typically used for approximate string matching [Chang and Lawler, 1994]. For the non-greedy RLZ algorithm, we require matching-statistics of each sequence \(T_i\) in a dataset of DNA sequences \(T\), where \(2 \leq i \leq r\), with respect to sequence \(T_1\).

Chang and Lawler described an algorithm to calculate \(M(X|Y)\), where \(X\) is of length \(m\) and \(Y\) is of length \(n\), in \(O(m + n)\) time. A suffix tree of \(Y\) is used for this purpose because of the ease of implementing suffix links. Recall from Section 3.1.3 that suffix links permits constant time access to the \((k + 1)\)th suffix from the \(k\)th suffix. First the algorithm finds the longest prefix of \(X[0 \ldots |X|]\) in the suffix tree of \(Y\). Let the position and length corresponding to this match be \((p_0, \ell_0)\) and let this suffix be the \(p_0\)th suffix of \(Y\). If \(\ell_0 > 1\), then the substring \(X[1 \ldots \ell_0]\) must also be present in \(Y\). In fact, this corresponds to the \((p_0 + 1)\)th suffix of \(Y\), which can be accessed via a suffix link. This ensures that longest prefix matching can continue from position \(\ell_0 - 1\) of the \((p_0 + 1)\)th suffix of \(Y\), without requiring to check the existence of the substring \(X[1 \ldots \ell_0]\) in \(Y\). The use of suffix links continues until all the longest matching prefixes are found.

Although suffix link implementations are simpler when using suffix trees, the data structure is not efficient to store compared to a suffix array. Implementing suffix links in suffix arrays is also non-trivial. Abouelhoda et al. [2004] proposed an enhanced suffix array that adds some features of a suffix tree to a suffix array, that also allows a matching-statistic to be calculated in \(O(m + n)\) time. Maaß [2006] proposed a simpler more space-efficient extension to suffix arrays to add suffix links.
Time complexity | Recall that the time complexity of the greedy RLZ algorithm is \( O(N \log n_1) \) for a collection \( T \) of \( r \) sequences containing \( N \) bases. The non-greedy algorithm also requires \( O(\log n_1) \) time to binary search the suffix array of \( T_1 \) at each position of the sequences. The matching statistics of each sequence \( T_i \), where \( 2 \leq i \leq r \), with respect to \( T_1 \) can be calculated in \( O(N) \) time. Therefore the time complexity of the non-greedy parsing of RLZ is \( O(N + N \log n_1) = O(N \log n_1) \).

Space complexity | The greedy RLZ algorithm used \( n_1(\log |\sigma'| + \log n_1) + n_{i'} \log |\sigma'| \) bits of space during compression where \( i' \) is the longest sequence in \( T \) that is not the reference. The non-greedy RLZ algorithm requires two additional arrays, the inverse suffix array and the LCP array, each requiring \( n_1 \log n_1 \) bits of storage. Therefore, the space used during non-greedy RLZ is \( n_1(\log |\sigma'| + 3 \log n_1) + n_{i'} \log |\sigma'| \) bits.

Decompression | The decompression algorithm is the same for both greedy and non-greedy versions of RLZ. Therefore the time and space complexity are unchanged.

Mutation hypothesis | Figure 6.4 showed that, when a factor is introduced to cover a single point mutation, that factor also covers some part of the region beyond the mutation. For example, in sequence \( T_2 \), a factor of length three is used to cover the mutation at the fourth position. Had this sequence been factorised by the non-greedy RLZ algorithm, then the factor covering the mutation will be of length one, and the subsequent factor will be of length ten, corresponding to the alignment between \( T_1 \) and \( T_2 \). We hypothesise that non-greedy RLZ will output shorter factors to cover mutations that span a few nucleotides, while the longer factors chosen by looking ahead are likely to cover regions of alignment between the sequence being compressed and the reference. In Section 6.2.4 we attempt to confirm this hypothesis, and if it is confirmed, as well as being a compression algorithm that produces biologically meaningful output, the non-greedy RLZ algorithm can also potentially be used as an alignment tool.

Next, we discuss some encoding techniques for short factors that improves the compressed sizes produced by both greedy and non-greedy versions of RLZ.

6.2.2 Efficient short factor encoding

The second modification to RLZ we propose is to encode the shorter factors using less bits. Recall that the position components of factors were encoded using \( \lceil \log n_1 \rceil \) bits each. Each length component is Golomb coded using a divisor \( b \), which requires at least \( \log b + 1 \) bits. Clearly, for short factors spanning only a few nucleotides, using at least \( \lceil \log n_1 \rceil + \log b + 1 \) bits is not efficient. For factors with a length of at most \( \ell_M \), it may be more efficient to encode the substring itself, provided the threshold length \( \ell_M \) is chosen appropriately. Such an encoding will also reduce cache misses.
during decompression, since the reference does not need to be accessed. From here on, the term short factor is used to refer to a factor with a length of at most $\ell_M$.

We adopted the simplest option of using a fixed-length encoder, and encoded the substring represented by a short factor using 2 bits per base. The length component of the factor also need to be encoded to allow the decompressor to determine how many 2 bit quantities are to be decoded. Other options were to use either Huffman or arithmetic coding to encode the substrings represented by the short factors. We discuss the possibility of using these alternatives as part of future work in Section 6.3.

A drawback of short factor encoding is determining the threshold length $\ell_M$ which makes it beneficial to use short factor encoding over the standard factor encoding. To avoid specifying a threshold length for the encoding decision, we compare the number of bits required to encode a factor as a short factor to the number of bits required to encode a factor as a standard factor, and choose the type that uses the least number of bits on a per factor basis.

Since two different encoders are used, the decompressor needs to distinguish between the two types. Therefore, an extra bit is included at the beginning of a factor to indicate the encoding type. A 0-bit indicates that the next factor is a short factor, while a 1-bit indicates a standard factor. A short factor is encoded by first Golomb coding the length (using a divisor that efficiently encodes small integers), then the substring represented by the factor is encoded. A standard factor is encoded by first encoding the position using $\log n_1$ bits, followed by the Golomb coded length.

In general, short factor encoding does not add any complexity to the RLZ compression or decompression algorithms. When encoding or decoding, a decision needs to be made about encoding or decoding a short factor or a standard factor, respectively. The complexity of this is $O(Z)$ for both the compressor and decompressor, where $Z$ is the total number of factors. Also, in the worst case, all the $N-n_1$ symbols in an input collection $T$ will belong to short factors, hence all $N-n_1$ symbols need to be accessed again to be encoded, requiring $O(N)$ time. However, the larger cost is in the factorisation itself which is $O(N \log n_1)$ for greedy and non-greedy versions of RLZ. Therefore, the $O(N)$ time to encode the $N-n_1$ symbols is not significant.

One aim of adopting short factor encoding is to ensure that if a set of sequences being compressed by RLZ is not related to the reference, the compressed size of these sequences would be no worse than using the na"ive 2 bpb representation. If the sequences in a collection are unrelated to the reference, then most factors are likely to be short factors. Using either the fixed-length 2 bpb encoder or an arithmetic encoder for these short factors will satisfy this aim.

The other aim of short factor encoding is to take advantage of non-greedy Lempel-Ziv factorisation. Recall that non-greedy factorisation is likely to introduce many short factors and to realise the compression gains, shorter factors need to be encoded differently to longer factors. In Section 6.2.4, we show the improvements in compression for RLZ using non-greedy factorisation in combination with short factor
6.2. OPTIMISED RELATIVE LEMPEL-ZIV COMPRESSION

... 10030697 10
16287 23
10086342 13
8689589 13
16336 48
3831041 11
16395 28
9166835 12
11588317 13
16448 84
787019 13
...

Figure 6.9: Some factors from sequence 273614N of the Scere collection.

encoding. Short factor encoding also improves compression in combination with the basic greedy RLZ implementation, although the improvements are not as significant as for non-greedy RLZ, as will be shown by the experimental results.

Use of short factor encoding will also assist in identifying the mutation regions of a sequence with respect to the reference, if the two sequences are homologous. In Section 6.3, we analyse short factors to determine if they cover regions of sequences that are known to contain mutations. Next we discuss the third RLZ optimisation.

6.2.3 Efficient factor position encoding

The final modification to RLZ we introduce is a better encoding technique for position components. The modification is based on the hypothesis that the position components of factors covering homologous regions of the sequence are predictable. In general, RLZ does not restrict where factors can originate from. This allows RLZ to compress sequences containing large-scale structural changes. The disadvantage is that position components require $\log_2 n_1$ bits each to encode.

We now describe a property of RLZ factorisation that allows the position component of factors to be predictable for certain collections. When compressing collections containing genomes from the same species, each sequence in the collection is homologous to most or all of the remaining sequences. Therefore, when a sequence from such a collection is factorised with respect to the chosen reference, the regions of the sequence that align to the reference tends to be covered by relatively long factors, while the regions of the sequence containing mutations tend to be covered by short factors. This behaviour was shown in our discussion of the effects of mutations on RLZ factorisation earlier in Section 6.1.2. When the factors from basic RLZ for the sequence 273614N of the Scere collection were examined, the series of factors consisted of alternating short and long factors, as shown in Figure 6.9.
The factors in bold font are the relatively longer factors, while the remaining factors are the shorter interleaving factors. Another point of interest in this example is that the \( j \)th long factor always has a position value that is less than the position value of the \( (j + 1) \)th long factor. In fact, in this example, the position of the \( (j + 1) \)th long factor is exactly \( x \) more than the position of the \( j \)th long factor, where \( x \) is the sum of the length of the \( j \)th long factor and the lengths of the smaller factors in between the \( j \)th and \( (j + 1) \)th long factors. For example, the second long factor position 16336 is equal to 16287+23+13+13, which is the sum of the position of the first long factor and the lengths of factors until the second long factor. The first long factor is likely to be representing a region of alignment between sequence 273614N and the reference, then a mutation is encountered, which is represented using the two factors (10086342,13) and (8689589,13), after which the alignment to the reference continues with the next long factor (16336,48).

The property of interest here is that the position components of the long factors form an increasing subsequence of integers, and the \( (j + 1) \)th integer is predictable from the \( j \)th integer. The position components of these long factors form a *longest increasing subsequence* or an LISS, which is the longest subsequence of a given sequence of items, where the items that belong to the subsequence are in increasing sorted order. We used the algorithm by Schensted [1961] that calculates the LISS of the sequence of position components in \( O(Z \log Z) \) time for the \( Z \) factors in the Scere collection. Around half the factors from this collection belong to the LISS and these factors tend to be long. The factors that did not belong to the LISS tend to be much shorter. From here on, we identify the factors in the LISS as LISS factors and the factors that are not part of the LISS as non-LISS factors. Figure 6.10a shows the length distribution of factors from the Scere collection that belong to the LISS and non-LISS factor classes. Most LISS factors have a length greater than 30, while most non-LISS factors tend to be shorter, usually having a length of around 9–16.

The difference in the length of the factors that belong to the LISS and non-LISS factor classes is even more prominent when non-greedy factorisation is used. Figure 6.10b shows the length distribution of factors from the Scere collection when the non-greedy RLZ variant that finds the local maximum factors was used. Then a much larger proportion of the non-LISS factors have a length of one, while most of the remaining non-LISS factors have a length less than 16. Most LISS factors still have lengths of above 30. The single nucleotide factors most likely represent SNPs. We will attempt to confirm this fact in the experimental section. The difference in factor lengths may confirm our earlier hypothesis that non-greedy factorisation allows us to distinguish between regions of the genomes that contain mutations from the regions of the genomes that align to the reference.

Since the LISS factors form a significant portion of the total number of factors in a collection, and the position components of these factors are predictable, we attempt to encode these positions more efficiently. Previously, we output each factor to disk
6.2. OPTIMISED RELATIVE LEMPEL-ZIV COMPRESSION

(a) Greedy factorisation

Figure 6.10: Factor length distribution for the Scere collection when the greedy and non-greedy relative Lempel-Ziv factorisations are used.

as soon as the factor was produced. However, since we now need to determine which of the factors belong to the LISS class and which do not, we store all factors $F_i$ for a sequence $T_i$ in memory. We use the algorithm by Schensted [1961] to obtain a bit vector $L_i$ of length $z_i = |F_i|$ that indicates which of the factors belong to the LISS. The first step of the encoding process is to write this bit vector to disk so that the decoder is aware of which factors are encoded as LISS factors.

One way to encode the positions of the LISS factors is to store the position $p_k$ of the $k$th LISS factor as the difference between the position value of the $k$th and $(k-1)$th factor, or $\delta(p_k, p_{k-1})$. This is possible because the condition $p_k > p_{k-1}$ is always satisfied given the definition of the longest increasing subsequence. Then, all non-LISS factors and the first LISS factor are encoded in the standard manner, while the remaining LISS factors are encoded using the position differences. The lengths are Golomb coded as usual. A drawback of this solution is that the position difference may still be relatively large, since an LISS factor is likely to be long, and
could potentially consist of thousands of bases.

A better solution is to use the length of an LISS factor and the lengths of the intermediate factors to predict the position of the next LISS factor. Then the position of the next LISS factor can be encoded as the difference between the predicted and actual position. Let the \((k-1)\)th LISS factor have a position \(p_{k-1}\) and a length \(\ell_{k-1}\). Also, let the \(k\)th LISS factor have a position \(p_k\) and a length \(\ell_k\). Let \(j_{k-1}\) and \(j_k\) be the locations of the \((k-1)\)th and \(k\)th factors in \(F^i\), respectively. We can predict the position of the \(k\)th factor as \(p'_k = p_{k-1} + \sum_{l=j_{k-1}}^{j_k-1} \ell_l\). In other words, we add the lengths of all factors from factor \(j_{k-1}\) until factor \(j_k\) to the position of the previous LISS factor to predict the position of the next LISS factor. Then we encode the difference \(\delta(p_k, p'_k)\), which we expect to be a small value if sequence \(T_i\) is homologous to \(T_1\). We encode all non-LISS factors and the first LISS factor in the standard manner, and the remaining LISS factors are encoded as the difference between the predicted and actual position. The lengths are Golomb coded. To encode the LISS factor positions, we use the following encoding technique. If the predicted position \(p'_k\) is the same as the actual position \(p_k\), then a 0-bit is output. Otherwise, a 1-bit is output, followed by another 0-bit if \(p_k < p'_k\) or a 1-bit if \(p_k > p'_k\). In the latter case where \(p_k \neq p'_k\), the difference \(\delta(p_k, p'_k)\) is Golomb coded.

Although this modification adds an extra bit for each factor to indicate whether it is encoded as an LISS or non-LISS factor, the gain in compression should be more significant given that most factors belong to the LISS. Also short factor encoding can be combined with LISS factor encoding to gain further compression improvements. Since non-LISS factors tend to be short, especially when non-greedy RLZ is used, these factors can be short factor encoded. In Section 6.2.4 we experiment with combining the three modifications to show the effects on compression.

The most significant change to the RLZ algorithm resulting from the introduction of LISS factor encoding is the need to store the factors for each sequence to determine which factors belong to the LISS and which do not. If a sequence \(T_i\) has \(z_i\) factors, then \(O(z_i)\) extra space is required. Also \(O(z_i \log z_i)\) time is required to find the LISS factors. Instead of using the static algorithm by Schensted [1961], an online algorithm that dynamically calculates the LISS can also be used [Samuels and Steele, 1981] to avoid the need to store factors in memory.

LISS factor encoding has little impact on the RLZ decompression algorithm. First the bit vector that indicates which of the factors are encoded as LISS factors needs to be read. Decoding non-LISS factors proceeds in the same manner as before. To decode LISS factors, the algorithm keeps track of the position of the previous LISS factor and then adds the lengths of the intermediate factors to predict the next LISS factor position. Then the decoded position difference for the current LISS factor is added to the predicted position to obtain the actual position for the current LISS factor. This does not change the complexity of the decompression algorithm.

Finally, this modification is only suitable for compressing collections where the
sequences are homologous to the reference. If a sequence contains many mutations with respect to the reference, then most factors are not part of the LISS and no compression is achieved. An appropriate divisor should be used when Golomb coding the LISS position differences so that the algorithm performs at least as well as the standard encoding when unrelated sequences are compressed using this option.

This concludes the discussion of the modifications that were explored to improve the compression performance of RLZ. Next, we evaluate the performance of RLZ after adding each of these modifications to the standard RLZ implementation.

### 6.2.4 Experimental evaluation

We now experimentally evaluate the compression gains from adding each of the three modifications to the basic RLZ algorithm. We also evaluate the compression gains from combining these modifications. The test data and environment for the experiments are discussed prior to the experimental results.

**Test data and environment**

To evaluate the improved RLZ algorithm, the same collections used to evaluate the standard RLZ algorithm in Section 6.1.4 were used, namely, Ec01, Spara, Scere, Athal, Sson, and Hsap. The improved RLZ implementation will be analysed in terms of the same three performance criteria as in Section 6.1.4, namely compressed size, compression and decompression speed, and approximate maximum memory usage during compression. Refer to Section 6.1.4 for details on the measurements obtained for these criteria, and on how the suffix array is constructed, and the reference is compressed. Additionally, the LCP array was constructed using the lcpdc algorithm [Puglisi and Turpin, 2008].

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

**Results**

First we analyse the compression performance of RLZ when the three improvements are combined. The combinations are:

- Non-greedy factorisation (lookahead)
- Non-greedy factorisation with short factor encoding (lookahead+shortfac)
- Non-greedy factorisation with LISS factor encoding (lookahead+liss)
- Non-greedy factorisation with both LISS factor encoding and short factor encoding (lookahead+liss+shortfac)
We experimented with non-greedy factorisation with look-ahead limits ranging from 0–30, and the variant that searches for local maximum factors. The look-ahead limit of zero is equivalent to the standard greedy factorisation algorithm. The compressed sizes for using the various combinations of the improvements described above to compress the Ecol, and Scere collections are presented in Figure 6.11. The dotted horizontal line represents the compressed size for each collection using the standard greedy RLZ implementation, which is used as a base line.

If short factor encoding is introduced to the standard RLZ implementation, then compression tends to improve over the base line. This is expected, since mutation regions tend to be covered by shorter factors, which can be encoded more efficiently. Similarly, when LISS factor encoding is introduced to the standard RLZ implementation, compression improves. This is also expected, since the genomes from the individual strains are likely to be very similar to the reference genome and form alignments, which can be exploited to produce better compression results using LISS factor encoding. Combining both the encoding techniques naturally leads to even better compressed sizes, since the non-LISS factors tend to be shorter, and can be encoded more efficiently with short factor encoding.

For all three collections, non-greedy factorisation alone leads to a higher compressed size than the base line. This is expected, since the compression gains of using non-greedy factorisation can only be realised if factors are encoded differently based on their lengths. Interestingly, even without modifying the encoding techniques, as the look-ahead limit increases, the compressed size lowers and approaches the base line. The compressed size is slightly higher for the local maximum non-greedy factorisation algorithm, and this is expected since many short factors need to be encoded prior to encoding each local maximal factor.

When short factor encoding is combined with non-greedy factorisation, the compressed sizes are lower by a few megabytes, compared to just using non-greedy factorisation. This shows that factors need to be encoded differently based on their lengths to utilise the gains of non-greedy factorisation. Similarly, combining non-greedy factorisation with LISS factor encoding also improves the compressed size by a few megabytes compared to just using non-greedy factorisation. However, for Ecol, this combination led to worse results than the base line, most likely because the reference is not very similar to the remaining sequences in the collection.

When non-greedy factorisation is combined with both LISS and short factor encoding, the compression results improve significantly compared to the base line. For the Scere collection, the compressed size almost halved compared to the base line. Even though the improvements are not as pronounced for the Ecol collection, there is still an improvement of around 5 Mbytes. The results are expected, since the longer factors tend to be part of the longest-increasing subsequence, and hence are LISS encoded, while the shorter factors tend not to be part of this subsequence, and are likely to be encoded by the short factor encoder. Therefore, a majority of
6.2. OPTIMISED RELATIVE LEMPEL-ZIV COMPRESSION

Figure 6.11: The variation in compressed size (in Mbytes) of the Ecol and Scere collections for changes in the look-ahead limit, using the various encoding techniques.

Factors are likely to be encoded by one of the two improved encoders, resulting in the better performance. Moreover, using the local maximum non-greedy factorisation algorithm produces the best compression results for all three collections. When combined with the LISS and short factor encoders, the compression gains of looking ahead to find the longest possible factors are realised, where the longest factors are likely to be part of the longest-increasing subsequence, while the shorter factors covering the regions between long factors can be short factor encoded.

Figure 6.12 displays the variation in compression and decompression times for the three collections as the look-ahead limit is varied. Generally, the compression speed is faster when non-greedy factorisation is used. Part of the reason is that we have carefully implemented the looking ahead functionality so as not to increase the compression cost in practice. Also, since longer factors are detected when looking ahead, more of the sequence is likely to be covered at a given iteration of the fac-
torisation algorithm, reducing the number of steps required to cover the sequence. For the variant that detects local maximum factors, the compression speed is slower compared to the greedy RLZ implementation. When looking ahead indefinitely, it is possible that looking ahead needs to continue for hundreds to thousands of bases before it can be confirmed that a local maximum factor is found. The cost of the binary searches required to look ahead indefinitely seems to outweigh the speed gains of covering longer regions of the sequence.

Although the decompression speed seems to vary significantly, it is always faster with non-greedy factorisation compared to when greedy factorisation is used. The most likely reason behind this speed improvement is the reduction in cache misses. With looking ahead, longer factors are introduced, which increases the number of symbols decoded per access to the reference. Since we have also enabled short factor encoding, the symbols represented by a short factor are accessed directly from the compressed sequence without needing to access the reference.

Overall, the combination of non-greedy factorisation, specifically the variant that searches for local maximum factors, short factor encoding and LISS factor encoding, produced the best compression results for RLZ. The best results from our experiments are reported in Table 6.4 along with the base line result for comparison (RLZ-std is the standard implementation and RLZ-opt is the implementation that includes all three modifications, except for Sson, which only includes non-greedy factorisation and short factor encoding). For all the collections, the optimised RLZ algorithm produced better compression results than the basic RLZ algorithm. The most significant improvements are made to the results of the Spara and Scere collections, which are two of the most repetitive collections. The Ecol collection is not as repetitive, so the improvements are not as significant. The K12 strain of E. coli may not be a suitable reference for the collection. The Athal collection, which consists of assembled contigs, produced somewhat better results with the modifications. We did not expect the compression results to improve significantly for this collection with LISS factor encoding, since the contigs are unlikely to be ordered in a manner that will align to the reference genome. Surprisingly, using LISS encoding produced better compression than if LISS encoding was not used.

The collection for which we expected better compression was Hsap, where we expected the compression results to be almost halved. However, on closer inspection, the Venter, Korean and Chinese genomes had a collective compressed size of 107 Mbytes using RLZ-std, while the size was 61 Mbytes using RLZ-opt, which is almost a halving of the compressed size. As more genomes are added to the collection, the improvements in the compression results will become evident.

Overall, the compression speed is slightly slower for RLZ-opt as expected, while the decompression speed tends to be faster. The memory usage tends to be 2–3 times greater for RLZ-opt compared to RLZ-std. This is expected due to the extra data structures required to simulate suffix links.
When the RLZ-opt results are compared to those of XMCompress from Table 6.3, the compressed sizes are now close to that of XMCompress, especially for Spara, Scere and Athal. The compression and decompression speeds of RLZ-opt are still much faster than for XMCompress. When the compression results of RLZ-opt are compared to 7-Zip, Sequitur and Re-pair, for the larger collections, the results are close to those achieved by these best performing general-purpose compressors, as shown in Figure 6.13. Even with the reduction in compression speed, RLZ-opt is still faster to compress than these compressors, while the decompression speed also remains competitive. Memory usage is also significantly lower.

We also attempted to confirm whether the shorter factors created by RLZ, specifically the non-LISS factors, encode mutations, and the LISS factors form the alignments between the reference and the sequence being compressed. To test this
Table 6.4: RLZ compression results for the collections in \texttt{dataset-rep}. The columns are: collection name, algorithm used, compressed size in Mbytes, average number of bits per base used, compression time in seconds, approximate maximum memory used during compression in Mbytes, and decompression time in seconds, respectively. The first line for each collection contains the results for RLZ prior to making the improvements. The second line contains the results for RLZ with all the improvements.

<table>
<thead>
<tr>
<th>Coll.</th>
<th>Algorithm</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decomp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>RLZ-std</td>
<td>23.14</td>
<td>1.18</td>
<td>124</td>
<td>31.72</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>18.41</td>
<td>0.94</td>
<td>142</td>
<td>64.48</td>
<td>3</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ-std</td>
<td>22.29</td>
<td>0.44</td>
<td>174</td>
<td>55.09</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>12.82</td>
<td>0.25</td>
<td>242</td>
<td>136.55</td>
<td>3</td>
</tr>
<tr>
<td>Scere</td>
<td>RLZ-std</td>
<td>17.06</td>
<td>0.29</td>
<td>141</td>
<td>57.13</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>8.90</td>
<td>0.15</td>
<td>229</td>
<td>131.06</td>
<td>3</td>
</tr>
<tr>
<td>Athal</td>
<td>RLZ-std</td>
<td>40.21</td>
<td>0.67</td>
<td>500</td>
<td>474.68</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>33.35</td>
<td>0.55</td>
<td>653</td>
<td>1,258.51</td>
<td>6</td>
</tr>
<tr>
<td>Sson</td>
<td>RLZ-std</td>
<td>20.18</td>
<td>0.17</td>
<td>173</td>
<td>48.02</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>19.08</td>
<td>0.16</td>
<td>328</td>
<td>103.24</td>
<td>5</td>
</tr>
<tr>
<td>Hsap</td>
<td>RLZ-std</td>
<td>716.05</td>
<td>0.50</td>
<td>10,084</td>
<td>1,023.03</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>RLZ-opt</td>
<td>670.42</td>
<td>0.47</td>
<td>14,488</td>
<td>2,701.61</td>
<td>124</td>
</tr>
</tbody>
</table>

Figure 6.13: Comparison of the compressed sizes (in bpb) for the test collections.

hypothesis, we compressed the chromosome 10 sequence of the 273614N strain of \textit{S. cerevisiae} against the chromosome 10 of the reference genome of the species. We analysed the mutations in the \texttt{ima4p} gene region of the sequence, which can be found in the position range 21054–22290 of the 273614N sequence. The region
aligns to the range 16766–18003 of the reference. All single nucleotide factors in this region correspond to SNPs that are confirmed by the SNP data included with the assembled genomes. In the same manner, the LISS factors in this region correspond to the alignment between the sequence and the reference. Two of the short factors in the region contain more than one nucleotide. One is a short substring `ctta`, where the first and last nucleotides correspond to SNPs. The second short substring `acagtcttca`, similarly contains two SNPs at the beginning and end of the substring. These are examples of two SNPs being encoded by one factor, since they were in close proximity (Section 6.1.2). We also used blastn [Zhang et al., 2000] to align sequence 273614N to the strain S288C, which is the reference genome used by the tool. Briefly comparing the factors to the alignment produced by blastn, generally, the insertions are covered by one or more factors and deletions break the alignment, as predicted. However, there are special cases where indels are in close proximity to point mutations, where factorisation does not occur as predicted.

We also tested the ability of RLZ to compress read sets. We used a collection of reads with 100 and 102 read lengths from the K12 strain of E. coli, produced by an Illumina GAIIx sequencer, for this experiment. The read set consists of 22,720,100 reads of length 100, and 22,696,116 reads of length 102, amounting to a total size of 4,587,013,832 bases. We used the K12 genome as the reference. The reverse complement of the genome was concatenated to the reference, since the RLZ algorithm is not implemented to detect reverse complement matches, and typically half the reads in a read set tend to originate from the reverse strand. Only non-greedy factorisation and short factor encoding options were enabled, since LISS encoding is not suitable for this dataset. The read set was compressed by RLZ to 222.27 Mbytes or 0.42 bpb, excluding the compressed reference. We also compressed the same read set with 7-Zip and it was only able to compress the collection to 319.80 Mbytes or 0.58 bpb. Therefore, RLZ is also suitable for read compression.

The experimental results confirm that the three optimisations to RLZ results in better compressed sizes and faster decompression, with only a small impact on the compression speed. Next we outline future improvements that can be made to RLZ.

6.3 Discussion

The standard RLZ algorithm, which uses a greedy Lempel-Ziv factorisation of a sequence with respect to a reference, is a simple algorithm that achieves excellent compression results for collections of individual genomes from the same species. This makes RLZ highly suitable for compressing the output of typical resequencing projects. After including three improvements, non-greedy factorisation, short factor encoding and efficient encoding of position components for homologous sequences, the compression performance improved even further, as was seen in the experimental results. For some collections, the compressed sizes were almost halved.
The three improvements also have biological significance. Non-greedy factorisation creates factors that preserve the boundaries between mutation regions and regions that are shared with the reference. Essentially, non-greedy factorisation forms an alignment between the genome being compressed and the reference. Short factor encoding tends to compress factors that cover the mutation regions, while the LISS encoder compresses factors that form the alignments between two genomes. RLZ could potentially be used as a fast method for measuring sequence similarity, and it could even be used to obtain a fast alignment between two sequences. However, the reliability of such a method is unclear.

Even though RLZ is restricted to compressing collections for which an appropriate reference is available, it is not just restricted to compressing complete genomes. As shown in the experiments, RLZ can also compress assembled contigs and read sets, given the availability of a reference genome. Therefore, RLZ is highly suitable for compressing the several types of data produced by sequencing projects. RLZ can also potentially be extended to perform read alignment in a reference genome, and could be used to determine whether reads contain errors. It may also be possible to apply RLZ in the context of metagenomics, especially if some reference species are known so that short reads or assembled contigs can be compressed against the references to determine their similarity levels. However, if there is little knowledge of the species in a sample, then RLZ will not be suitable, since a reference is required.

We now consider some improvements that can be made to RLZ to improve the compression speed and the compressed size. In our implementation, we used a basic suffix array for both greedy and non-greedy factorisation. Two additional data structures were also required for non-greedy factorisation, increasing the memory use to $3n \log n$ bits. Had a compressed suffix tree been used for this purpose, a substring $P$ can be searched for in $O(|P|)$ time, instead of $O(|P| \log n)$ time using a suffix array. However, as discussed earlier, the current implementations of compressed suffix trees and arrays reduce the RLZ compression speed significantly, even though the memory usage of RLZ improves significantly as a result. We believe that the fast compression speed of RLZ is far more important relative to its memory usage, hence chose not to use these compressed data structures. However, if faster compressed suffix trees or arrays are introduced in the future, it may be beneficial to incorporate these implementations into RLZ.

An alternative method of storing the reference and its suffix array is to store the reference as an FM-Index. Recall from Section 3.4.1 that an FM-Index stores the Burrows-Wheeler transform (BWT) of a string $S$ and some other data structures to implement count(), locate() and display() queries. Ferragina and Manzini noted the interchangeability between the BWT of a string and its suffix array. This is the property that allows the reference to be stored as an FM-Index. In practice, there are several disadvantages to using the FM-Index representation of $T_1$, $FM(T_1)$, as opposed to the standard representation. Since single DNA sequences are not very
6.3. DISCUSSION

repetitive, the difference between using $n_1 H_k(T_1)$ bits (the size of the FM-Index) and $n_1 \log |\sigma'|$ bits will not be very significant. Note that the suffix array is only required during compression so it is not added to the compressed data. Moreover, decompression using an FM-Index representation is slower because a substring cannot be accessed from $FM(T_1)$ as directly as accessing a substring from $T_1$. Retrieving a substring of $l$ symbols from $FM(T_1)$ requires $O(l \log |\sigma'| + \log^{1+\epsilon} n)$ compared to $O(l)$ time to retrieve the same substring from $T_1$. The $T_1$ representation also has better memory locality than $FM(T_1)$. To test its practicality, we implemented a variant of RLZ that uses the FM-Index, and used it to compress and decompress the Spara collection. In terms of the compression time, this method faster by a few seconds. However, as predicted, the decompression time was many orders of magnitude slower (over 5 mins to decompress compared to the original 4 secs), since it is less efficient to access substrings from $FM(T_1)$ as it is to access from $T_1$. Interestingly, the compressed size was also larger by a few megabytes as the $FM(T_1)$ was larger than the reference compressed with 7-Zip. The $FM(T_1)$ also occupied a few megabytes more memory than the reference stored using $n_1 \log |\sigma'|$ bits.

Another feature not included in RLZ is reverse complement repeat detection. Most DNA compressors include reverse complement detection, since DNA sequences tend to contain these repeats as discussed in Section 2.1.4. We attempted to implement this feature by concatenating the reverse complement reference to the end of the reference, so that reverse complement repeats can be detected without modifying the original RLZ factorisation algorithm. This doubles the size of the reference and the suffix array, and reduces the compression speed due to the need to search in a longer string. However, the improvements to the compression results for our test collections were not justified given the increased memory usage and reference length. However, note that for reads and contigs, such as the Sson collection, reverse complement repeat detection significantly improves compression. It may be advantageous to modify RLZ to detect reverse complements without increasing the size of the reference and the suffix array.

Improvements can also be made to the encoding techniques, specifically, for short factor encoding. The current implementation uses 2 bp to encode each nucleotide that is part of a short factor. Instead, Huffman coding or arithmetic coding can be used. However, Huffman coding may not be appropriate, since the probabilities of symbols or groups of symbols are unknown in advance. These probabilities are dependent on the collections being compressed so an extra pass through the collection will be necessary to obtain accurate probability distributions. Adaptive Huffman coders exist but they are not very efficient [Witten et al., 1999]. Arithmetic coding is much more suitable, since the probability distribution can be modified as more symbols are read. We leave this modification as a future exercise.

During the writing of this thesis Deorowicz and Grabowski [2011] improved on the RLZ algorithm in several ways, which resulted in significantly better compressed
sizes, as well as faster compression speeds. We discuss their improvements below.

**Comparison of RLZ to the GDC algorithm**

The GDC algorithm [Deorowicz and Grabowski, 2011] is an improved version of RLZ. Firstly, the reference string $T_1$ is divided into blocks, and triplets of symbols in each block are byte-packed and then Huffman coded. The model for Huffman encoding is shared between the blocks. Secondly, the matching substrings between each sequence and the reference are found using hash values in the traditional manner for LZ77 compressors, instead of using a suffix array as in RLZ. Therefore, the hash values of all overlapping substrings of length $M_1$ are also stored. For smaller genomes, the $M_1$ value used by the authors is 13, while for larger genomes like the Hsap genomes, an $M_1$ value of 20 is used to minimise collisions. It is unclear how the hash values are stored without significantly increasing memory usage. Our experiments confirm that the memory usage of GDC is less than that of RLZ, and this is most likely because the hash values are only stored for the reference rather than for the entire collection.

Then each sequence $T_i$, where $2 \leq i \leq r$, is factorised by finding matching substrings in $T_1$ using the hash values. GDC encodes a factor as the difference between the position of occurrence in the reference and the current sequence, and the length of the match. The position difference and the length are variable-length encoded for storage. In hindsight, it is more efficient to encode the offset of the position difference rather than the position of occurrence with $\log n_1$ bits. More often than not, this offset will be significantly less than $n_1$.

The non-greedy parsing decision of GDC is also different to RLZ. GDC chooses a shorter matching substring over a longer match if the offset from the reference for the shorter match is cheaper to encode. This is essential given the type of encoding used. While RLZ handles approximate matches with LISS factor encoding, GDC explicitly permits up to $k$ mismatches when finding substring matches. Limits are placed on the minimum number of exact matches that must occur between gaps to ensure that the approximate matches found are legitimate, and to reduce the encoding cost. Substrings not occurring in the reference, and mismatches are byte-packed and Huffman encoded, similar to the reference. Unlike RLZ, GDC also accepts DNA sequences containing any symbol from the extended DNA alphabet, and the input is expected in the commonly used FASTA format.

The combination of using the compressed reference, and finding substring matches near constant time using a hash table, makes GDC faster. Also the combination of the improved encoding techniques, and the modified parsing algorithm allows GDC to produce better compression results. The experimental results presented by Deorowicz and Grabowski [2011] use several variants of the GDC algorithm, some of which improve the quality of the reference. We discuss these improvements in Chapter 8. The two variants of relevance to this chapter is GDC-normal and GDC-fast. The
former is the algorithm described above, the latter is a variant that does not Huffman code the reference to improve access speeds. Overall, the results do not show much of a difference in the compression speed between GDC-normal and GDC-fast, but the compression results are slightly better when using GDC-normal. We ran our own experiments with GDC-normal on our test collections from DATASET-REP, to compare with RLZ. Default parameter values were used for GDC, except for the Hsap genome, where the value $M_1 = 20$ was used to reduce hash collisions and speed up compression. The memory usage was measured using the Unix command `ps v`, since valgrind was not able to run on the GDC executable.

In Table 6.5, we present a comparison between the results of GDC-normal and RLZ-opt. For the three collections of fully assembled genomes, GDC outperformed RLZ, both in terms of compressed size and compression speed. The compression speed is particularly impressive, since GDC tends to be a few times faster than RLZ. For the Athalina and Sson contig collections, the compression performance of GDC is not significantly different to RLZ. GDC marginally outperformed RLZ for the Athal collection, while RLZ outperformed GDC for the Sson collection. As for RLZ, we appended the reverse complement of the Sson reference genome to itself in order to achieve better compression. For the Hsap collection, GDC also does not perform as well as RLZ but the compression speed is much faster. Compared to RLZ, the GDC-compressed reference is slightly larger, but the compressed collection is slightly smaller. Deorowicz and Grabowski [2011] also compressed a collection of 70 human genomes, which contain 218,962 Mbases. They claim that RLZ-opt compressed this collection to 1,651 Mbytes, while GDC compressed this collection to 1,146 Mbytes. This is unsurprising, since GDC can compress the collections to a smaller size than RLZ. More importantly, GDC compressed this large collection in less than 2 hours on their test system, while RLZ required 45 hours on their system. In terms of the decompression speed, GDC tends to be slightly slower than RLZ. Decompressing symbols from a Huffman encoded reference is slower than having direct access to the symbols in the reference. Memory usage of GDC is mostly similar to RLZ.

Overall, the authors of GDC have made several key improvements to the RLZ algorithm to improve the compressed size and the compression speed. As we see later in Chapter 8, several other improvements have been made by the authors to further improve the compression performance of GDC.

**Selecting a reference**

An issue of importance not discussed so far is the selection of a reference. So far we have assumed that a collection being compressed contains an appropriate reference. However, this may always not be the case. An example is the Eco1 collection, which did not contain a reference genome for the species, hence was not very compressible with RLZ. Even the reference genome for a species may not necessarily be the se-
Table 6.5: RLZ and GDC compression results for collections in dataset-rep. The columns are: collection name, algorithm used, compressed size in Mbytes, average bits per base used, compression time in seconds, approximate maximum memory usage during compression in Mbytes, and decompression time in seconds, respectively.

<table>
<thead>
<tr>
<th>Coll.</th>
<th>Algorithm</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decomp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>RLZ-opt</td>
<td>18.41</td>
<td>0.94</td>
<td>142</td>
<td>64.48</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>13.72</td>
<td>0.70</td>
<td>64</td>
<td>80.45</td>
<td>3</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ-opt</td>
<td>12.82</td>
<td>0.25</td>
<td>242</td>
<td>136.55</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>8.77</td>
<td>0.17</td>
<td>24</td>
<td>136.28</td>
<td>4</td>
</tr>
<tr>
<td>Scere</td>
<td>RLZ-opt</td>
<td>8.90</td>
<td>0.15</td>
<td>229</td>
<td>131.06</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>6.55</td>
<td>0.11</td>
<td>15</td>
<td>132.25</td>
<td>4</td>
</tr>
<tr>
<td>Athal</td>
<td>RLZ-opt</td>
<td>33.35</td>
<td>0.55</td>
<td>653</td>
<td>1,258.51</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>32.75</td>
<td>0.54</td>
<td>251</td>
<td>1,034.50</td>
<td>7</td>
</tr>
<tr>
<td>Sson</td>
<td>RLZ-opt</td>
<td>19.08</td>
<td>0.16</td>
<td>328</td>
<td>103.24</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>23.80</td>
<td>0.21</td>
<td>216</td>
<td>140.04</td>
<td>8</td>
</tr>
<tr>
<td>Hsap</td>
<td>RLZ-opt</td>
<td>670.42</td>
<td>0.47</td>
<td>14,488</td>
<td>2,701.61</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>719.74</td>
<td>0.50</td>
<td>4,675</td>
<td>2,243.08</td>
<td>134</td>
</tr>
</tbody>
</table>

In this chapter, we focused on compressing collections that are suitable for RLZ compression, and did not explore the consequences of compressing collections that are unsuitable for relative compression. As we will see in Chapter 8, the results for compressing a collection like Hemo or Inf1 from dataset-rep are not very promising, since the sequences in these collections cannot be represented by a single reference. The results of compressing such collections tend to be worse than a naïve encoding. To make RLZ more suitable for compressing general sets of DNA sequences, we explore some dictionary construction techniques in Chapter 8.

### 6.4 Chapter summary

In this chapter we introduced a relative compression algorithm called RLZ to compress sequence collections produced by resequencing projects. RLZ is suitable for compressing multiple genomes from the same species, or more generally, collections of homologous sequences. For such collections, RLZ produces good compression results with fast compression and decompression speeds, and low memory usage, as
verified by the experimental results. These properties make the algorithm scalable for compressing the large collections commonly produced by sequencing projects.

The basic RLZ algorithm uses a greedy relative Lempel-Ziv parsing of each sequence in a collection with respect to a reference. The parsing produces a set of position and length pairs, called factors, which represent substrings shared between the sequences being compressed and the reference. Then a non-greedy relative Lempel-Ziv parser was introduced, along with better encoding techniques for shorter factors which are more common with non-greedy parsing. A better position encoder was also introduced for factors that form alignments to the reference. Combining all three modifications led to even better compression results than the standard algorithm. The three modifications are based on biological observations, which is the most likely reason behind the significant improvements observed in the results.

While being able to compress fully assembled sequences, we also showed that RLZ can be used to compress assembled contigs, as well as read sets. This makes RLZ useful for compressing different types of datasets output by sequencing projects.

One of the main advantages of RLZ is that the output of the compressor can be modified to construct a self-index, where display(), count() and locate() queries can be issued on the compressed index. In the next chapter, we present the necessary modifications to the compressed output of RLZ to construct a self-index, and the algorithms for implementing the display(), and versions of count() and locate() queries with some limitations.
CHAPTER 6. EFFICIENT GENOME STORAGE WITH RLZ
Chapter 7

The RLZ self-index*

Compression algorithm implementations such as Re-pair, Sequitur, dna-x and XMCompress produce excellent results when compressing highly repetitive DNA collections. However, as stated earlier, these results are only achieved by concatenating the individual files into a single file, often without delimiters to signal sequence boundaries. As a result, if only a single sequence or a substring of a sequence is to be extracted from the collection, the entire collection needs to decompressed. While efficient storage is an important factor, the ability to access the compressed collection may be a desirable feature if the collection needs to be accessed frequently. The RAY algorithm [Cannane and Williams, 2001] permits individual sequence extraction without decompressing the entire collection. Self-indexes not only permit individual sequence extraction, but also allow substring extraction from compressed sequences.

Another significant contribution made by self-indexing algorithms is the ability to perform simple search queries on a compressed collection [Navarro and Mäkinen, 2007]. The two basic queries are the count(P) and locate(P) queries, which return the number of occurrences of a substring P and the positions of occurrences, respectively. In Bioinformatics research, substring searching is essential for detection of sequences containing a given motif or mutation, and more generally for finding sequences in a collection that contain a substring of interest. Most self-indexes, such as the FM-Index, RLCSA, and LZ-End support these search queries, although it is more natural for the indexes based on suffix arrays to support these queries than for indexes based on LZ77 compression, for which it is necessary to include additional data structures alongside the compressed output to support the random access and search queries efficiently. Therefore, compression algorithms that support these queries tend to produce larger compressed output than standard compressors.

In this chapter, we extend RLZ to a self-index that supports random access and search queries with some limitations, on the compressed collection. The existing compressed output is modified and some new data structures are added to the compressed output to construct the self-index. The RLZ factors permit the random access

---

*An earlier version of the display() query algorithm was published in Kuruppu et al. [2010].
query to be implemented trivially with only small changes required to the existing compressed output. Therefore we first discuss the addition of the random access query. Although the compressed output is larger as a result of these modifications, our experimental analysis of the query performance shows that the RLZ random access query is faster than that of existing self-indexes that also support this type. On the other hand, search query support cannot be added trivially to RLZ compressed collections, and several new data structures need to be added to the compressed output, resulting in a significantly larger index size. A limitation of our algorithms is that we must make the assumption that substrings being searched for do not span over more than two factors. We experimentally evaluate several RLZ self-index variants with various index size and query time trade-offs. We also compare the query performance to that of existing self-indexes. The experimental results show that the current query performance is slow and impractical for evaluating large sets of queries. Therefore, significant improvements need to be made to the data structures that support the search queries to be competitive with existing self-indexes.

7.1 The display() query

First we describe the algorithm to invoke random access queries on a RLZ-compressed collection and the modifications required to the compressed output of RLZ to enable such a query to be invoked. The random access query is commonly known as the display() query, and it is formally defined for a collection $T$ of $r$ DNA sequences as follows:

**Definition 6.** Let $T_c$ be the RLZ compressed output for a collection $T$ of $r$ DNA sequences. Each sequence $T_i$ for $2 \leq i \leq r$ in $T_c$, is represented as $LZ(T_i|T_1) = (p^1_i, \ell^1_i)(p^2_i, \ell^2_i)\ldots(p^z_i, \ell^z_i)$, where $T_i$ is parsed into $z_i$ factors with respect to sequence $T_1$, and $p^j_i$ and $\ell^j_i$ are the position and length components of the $j$th factor. Then a $display(i,s,e)$ query returns the substring $T_i[s\ldots e]$ from the compressed collection $T_c$, where $1 \leq i \leq r$, $0 \leq s < n_i$ and $s < e \leq n_i$.

First, we describe the modifications to the compressed output of RLZ required to enable the display() query to be implemented on RLZ-compressed collections. Then we describe the display() query algorithm.

7.1.1 Modifications to RLZ-compressed output

Recall from Chapter 6 that RLZ takes a collection $T$ of $r$ DNA sequences, factorises each sequence $T_i$, where $2 \leq i \leq r$, with respect to the reference string $T_1$, then encodes the reference and the factors to produce the final compressed output $T_c$. The reference was compressed with 7-Zip, and the position component of each factor was encoded with $\log n_1$ bits each, while the length component of each factor was Golomb coded. In this section, we assume that the standard RLZ compression algorithm was used to compress the collection, as specified in Section 6.1.
A drawback of using a standard compressor to compress the reference is that, to extract a substring, the entire string needs to be decompressed. Since factors point to substrings occurring in the reference, to implement \textit{display()}, it is necessary to have fast access to these substrings. In other words, it is infeasible to decompress the reference in its entirety for every \textit{display()} query invoked. As a result, we no longer compress the reference, but instead store the reference by encoding each nucleotide using 3 bpb (recall that the DNA alphabet accepted by \textit{RLZ} is \(\sigma' = \{a, c, g, t, n\}\)). Although this naïve encoding does not compress the reference, it allows a substring of length \(\ell\) to be retrieved in \(O(\ell)\) time, which is an important property that is necessary to implement the \textit{display()} query efficiently. In Section 7.1.6, we also explore the possibility of storing the reference as an \textit{FM-Index}.

The position and length components of each factor also need to be accessed efficiently. In the standard \textit{RLZ} algorithm, the factors are stored separately for each sequence, and each factor is stored as a position followed by the length. Here, we modify this representation to store all the position components of factors separate from the length components. The position components of factors from all sequences are concatenated into an array \(P\), where each position component is encoded using \(\log n_1\) bits. This representation allows the \(j\)th factor of the collection to be retrieved in constant time. A data structure containing sequence boundaries is necessary to identify which factors belong to which sequence. The representation for this data structure is discussed shortly. The length components are not explicitly stored but can be calculated using the compressed bit vector described below.

To implement random access, it is necessary to efficiently determine which factor covers a given position of a sequence. This can be accomplished using a compressed bit vector that contains a 1 bit at every position where a factor begins. Then, given a position \(p\) for a sequence \(i\), a rank\(_{1}(p)\) query can be used to obtain the factor that contains position \(p\). Instead of constructing individual compressed bit vectors for each sequence, we construct a compressed bit vector for the concatenated bit vectors. Since this compressed bit vector can be used to retrieve the lengths of factors using select\(_{1}\) queries, we do not explicitly store the lengths. The compressed bit vector \(L\) is stored using the \textit{sdarray} representation by Okanohara and Sadakane [2007], since the 1-bits tend to be sparse, especially for collections where each sequence is similar to the reference. To determine the sequence boundaries, an additional array \(B\) of \(r\) elements containing the starting position of the first factor in \(L\) for each sequence is stored. Entry \(B[1]\) is ignored, since it represents the reference.

The four data structures, the reference string \(T_1\) stored using 3 bpb, the array \(P\) containing the position components of all the factors, the compressed bit vector of factor start positions \(L\) and the sequence boundary array \(B\), are the necessary components for invoking the \textit{display()} query on an \textit{RLZ}-compressed collection. Next we describe the \textit{display()} query algorithm.
7.1.2 The display() algorithm

We now describe the RLZ display() query algorithm. Let an RLZ-compressed collection \( T \) for a collection of \( r \) sequences contain the reference string \( T_1 \), factor positions \( P \), factor start positions in each sequence \( L \), and the sequence boundaries \( B \). When a \( display(i,s,e) \) query is invoked, where \( 2 \leq i \leq r \), \( 0 \leq s < n_i \) and \( s < e \leq n_i \), the algorithm returns the substring \( T_i[s \ldots e] \).

The first step of the algorithm is to determine the factor that contains position \( s \) of sequence \( T_i \). First, a \( j = \text{rank}_1(L, B[i] + s) - 1 \) query is used to determine the number of factors that occur at or before position \( B[i] + s \). Recall from the previous section that the \( L \) data structure contains a concatenated set of bit vectors, hence the position at which \( s \) occurs in \( L \) is determined using the sequence boundary array \( B \). The \( j \)th factor of the collection contains position \( s \). The position component of the \( j \)th factor can be retrieved as \( p_j = P[j] \). The length component of the factor needs to be retrieved from the compressed bit vector \( L \) as \( \ell_j = \text{select}_1(L,j + 1) - \text{select}_1(L,j) \). In other words, the length of the factor is the number of bits between the 1-bit of the \( j \)th factor and the 1-bit of the \((j + 1)\)th factor.

Before retrieval begins, the position at which to start decoding from needs to be determined, since position \( s \) could be anywhere within the factor. This offset can be calculated as \( o = B[i] + s - \text{select}_1(L,j) \), which is the distance of position \( B[i] + s \) from the start of the factor. The second step of the algorithm is to begin the nucleotide retrieval from \( T_1[p_j + o] \). If all \( e - s \) symbols can be retrieved from substring \( T_1[p_j + o \ldots p_j + o + \ell_j] \) then the display() query is answered. However, the substring \( T_i[s \ldots e] \) may span more than one factor. If this is the case, then the retrieval can continue from the \((j + 1)\)th factor. Let \( p_{j+1} = P[j + 1] \) and \( \ell_{j+1} = \text{select}_1(L,j + 2) - \text{select}_1(L,j + 1) \) be the position and length components of the \((j + 1)\)th factor. Then some or all of the remaining nucleotides of the query can be retrieved from \( T_1[p_{j+1} \ldots p_{j+1} + \ell_{j+1}] \). If all \( e - s \) nucleotides were not retrieved by this stage, then the algorithm continues to extract nucleotides from the \((j + 2)\)th factor, the \((j + 3)\)th factor and so on until all the \( e - s \) nucleotides are retrieved.

Algorithm 10 illustrates the display(i,s,e) algorithm for RLZ.

So far, the display() query algorithm for sequences other than the reference string was discussed. Implementing the display() query for the reference is trivial, since the substring can be directly accessed from \( T_1 \). Another special case not considered in Algorithm 10 is the factors that represent run-length encoded \( n \) nucleotides. This special case can be incorporated into the second step of the algorithm to output as many \( n \) nucleotides as necessary if the condition \( p == n_1 \) is satisfied. Next we analyse the time and space complexity of the display() algorithm.
7.1. THE DISPLAY() QUERY

Algorithm 10 RLZ\_display\((i, s, e)\) takes a sequence \(i\), a start position \(s\) and an end position \(e\), then returns the substring \(T_{[s \ldots e]}\) from the compressed collection.

1: \{Step 1: Get the factor to start retrieving from\}
2: \(j \leftarrow \text{rank}_1(L, B[i] + s) - 1\)
3: \(o \leftarrow B[i] + s - \text{select}_1(L, j)\)
4: \(\text{syms} \leftarrow 0\)
5: \textbf{while} \(\text{syms} < e - s\) \textbf{do} \{Step 2: Retrieve symbols from factors\}
6: \(p \leftarrow P[j] + o\)
7: \(\ell \leftarrow \min(\text{select}_1(L, j + 1) - \text{select}_1(L, j), e - s - \text{syms})\)
8: \textbf{output} \(T_1[p \ldots p + \ell]\)
9: \(\text{syms} \leftarrow \text{syms} + \ell\)
10: \(j \leftarrow j + 1, o \leftarrow 0\)
11: \textbf{end while}

7.1.3 Time and space complexity

First we discuss the space complexity of the modified output of the RLZ compression algorithm that enables the \textit{display()}\ query to be implemented on RLZ-compressed collections. Recall from Section 7.1.1 that the data structures required to implement \textit{display()} are the reference string \(T_1\), the array of factor positions \(P\), the compressed bit vector of factor start positions \(L\), and the array of cumulative sequence lengths \(B\). Let the compressed collection \(T_c\) contain \(Z\) factors.

Since the reference can contain nucleotides from the alphabet \(\sigma' = \{a, c, g, t, n\}\), we chose to store the reference string \(T_1\) using 3 bpb. This is to ensure that constant time access to retrieve any nucleotide from the string is possible. This representation requires \(3n_1\) bits of space. The concatenated array of the position components of all the \(Z\) factors is stored using \(\log n_1\) bits per position, requiring \(Z\log n_1\) bits overall. The compressed bit vector of factor start positions \(L\) containing \(N - n_1 \approx N\) bits is stored using the \texttt{sdarray} representation of Okanohara and Sadakane [2007], which requires \(Z \log \frac{N}{Z} + O(Z)\) bits of space. Finally, the cumulative sequence length array \(B\) contains \(r\) entries and each entry is stored using \(\log N\) bits, requiring \(r \log N\) bits in total. Overall, the space complexity of the modified RLZ compressed output is \(3n_1 + Z \log n_1 + Z \log \frac{N}{Z} + O(Z) + r \log N\) bits.

Recall from Section 6.1.2 that the RLZ output for the standard compression algorithm requires \(n_1H_k(T_1) + 2Z \log n_1\) bits of space, and even less space in practice, since Golomb coding is used to store the factor lengths. The \(H_k(T_1)\) term tends to amount to around 2 bits in practice due to single sequences being less compressible. Therefore, around \(n_1\) extra bits are required to store the reference in the modified compressed output. The space usage for positions is unchanged while the lengths tend to use more space than \(Z \log n_1\) bits. The \(r \log N\) bits needed to store the sequence boundaries is also not necessary for the standard compression algorithm.

Although the modifications require more storage space, the experimental results in Section 7.1.5 show that the extra cost is not very significant for repetitive collections.
We now analyse the time complexity of Algorithm 10. Recall from Section 3.1.4 that when using the \texttt{sdarray} representation, a \texttt{rank}_{1} query requires $O(\log \frac{N}{Z})$ time, while a \texttt{select}_{1} query requires $O(1)$ time. The algorithm requires a single \texttt{rank}_{1} query (line 2) to determine the initial factor to start retrieving nucleotides from. The remaining operations, such as determining the offset to start decoding from (line 3), extracting the position and length components of each factor (lines 6–7), and retrieving each nucleotide from $T_{1}$ (line 10), are constant time operations. Therefore, $O(e - s)$ time is required to retrieve $e - s$ nucleotides. The overall time complexity of a single \texttt{display()} query is $O(\log \frac{N}{Z} + e - s)$.

Comparing this result to the complexities of the \texttt{display()} queries implemented by \texttt{COMRAD}, \texttt{LZ-End}, \texttt{RLCSA} as discussed in Section 5.3.2, \texttt{RLZ} has a query performance that is close to the optimal $O(e - s)$ complexity that can be achieved by \texttt{LZ-End} under certain conditions. In practice, \texttt{RLZ} is likely to be faster, since most queries will require $O(e - s + h)$ time complexity for \texttt{LZ-End} [Kreft and Navarro, 2011]. The \texttt{RLZ display()} query will also be faster than those of \texttt{COMRAD} and \texttt{RLCSA}, since there is no overhead of recursively accessing non-terminals, or accessing a compressed suffix array. Comparing the space complexities is not trivial so we compare the space used in practice during the experimental evaluation in Section 7.1.5.

Recall that in the standard implementation, as soon as a factor is determined, it is output to disk. This is not possible for the modified output, since the compressed bit vector $L$ cannot be constructed dynamically and the sequence boundaries are only known once all the sequences are read. Hence all factors are stored in memory, and the compressed index is written to disk once all the sequences are factorised. Therefore, the modified algorithm consumes more memory during compression than the standard \texttt{RLZ} algorithm. We compare the \texttt{display()} query performance of \texttt{RLZ}, as well as the size of the index with other compressors that also implement this query in Section 7.1.5. Next we briefly discuss the random access implementation of \texttt{GDC}, which is the improved \texttt{RLZ} algorithm as discussed in Section 6.3.

### 7.1.4 GDC display() query

Like \texttt{RLZ}, the compressed output of \texttt{GDC} must also be modified to support the \texttt{display()} query. Firstly, each sequence in the collection being compressed is divided into approximately equal-sized blocks, and factors are not permitted to cross block boundaries. Secondly, the authors explored two alternatives to store the reference. One was to byte-pack the reference, similar to our approach of storing the reference using 3 bpb, except \texttt{GDC} supports the entire extended DNA alphabet. The other is to Huffman code groups of nucleotides. The first approach handles \texttt{display()} queries faster than the second approach, which requires an entire Huffman-coded block to be decoded to decode a factor. Optimisations such as caching recently accessed blocks were used to improve the query speed when the Huffman-coded reference is used.
For a \textit{display}(i,s,e) query, first the algorithm conducts two binary searches on the compressed sequence \textit{i} to find blocks that contain the start and end positions. Binary searching is necessary, since the compressed sequence blocks are not of equal lengths. The blocks are then decompressed to obtain the substring. Care is taken to skip over the prefix of the first block if position \textit{s} occurs after the beginning of the block, and the last block is only decompressed until position \textit{e} is reached.

We expect that the \textbf{RLZ} \textit{display()} query implementation will extract a substring faster than the \textbf{GDC} implementation. With \textbf{RLZ}, once the factor to start extracting from is obtained, the symbols can be accessed directly. In the \textbf{GDC} implementation, parts of the sequence containing the substring of interest and the necessary parts of the reference need to be decompressed. On the other hand, the \textbf{GDC} index is likely to be more compact than the \textbf{RLZ} index, since it is stored compressed, unlike the \textbf{RLZ} index, which stores the factors and the reference uncompressed. Next we experimentally evaluate the \textit{display()} query performance of \textbf{RLZ}.

7.1.5 Experimental evaluation

Here we experimentally evaluate the \textbf{RLZ} \textit{display()} query performance. Prior to the evaluation, we present the test collections and the experimental environment. We also compare the performance of the \textbf{RLZ} \textit{display()} query to the equivalent queries of some existing indexes in the same domain.

**Test data and environment**

We use the test collections and environment used to evaluate the \textit{display()} query performance of \textbf{COMRAD} in Section 5.3.3. We selected the highly repetitive \textit{Sson}, moderately repetitive \textit{Spara} and less repetitive \textit{Athal} collections from \textbf{DATASET-REP}. An \textbf{RLZ} index that supports the \textit{display()} query was built for each collection.

For each collection, we randomly generated 2000 queries each for the lengths, 10, 100, 1,000, 10,000 and 100,000. The first 1000 queries were used to ensure that all the necessary information is cached before the remaining 1000 queries are timed. The time to retrieve the symbols for a query was measured using the \texttt{gettimeofday()} function. The query time does not include the time taken to output the substring. We report the number of microseconds required to extract a symbol averaged over the latter 1000 queries for each length. The queries for each length were invoked five times for each collection, and the best result from the five runs was reported.

To compare the \textbf{RLZ} random access implementation to other compressed indexes that implement this query, we chose the \textbf{COMRAD}, \textbf{GDC}, \textbf{RLCSA} and \textbf{LZ-End} implementations. The \textbf{COMRAD} and \textbf{GDC} [Deorowicz and Grabowski, 2011] random access implementations were discussed in Sections 5.3 and 7.1.4, respectively. \textbf{RLCSA} [Mäkinen \textit{et al.}, 2009; Mäkinen \textit{et al.}, 2010] and \textbf{LZ-End} [Kreft and Navarro, 2010, 2011] were introduced in Sections 3.4.2 and 3.4.3, respectively. We built indexes for each col-
lection using each of the chosen compressed indexes. All compressors except LZ-End can build an index for multi-sequence collections. For LZ-End, the individual sequences were concatenated, and the queries were adjusted accordingly. The COMRAD parameters used were a minimum frequency threshold $F = 4$ and the initial substring length $L = 16$. GDC has many parameters but we chose the parameters that give the fastest retrieval speed, which are `-rm0 -bd8 -br8`, corresponding to storing an uncompressed reference, and reference and sequence block sizes being set to $2^8$.

The RLCSA parameters used were, a block size of 32, sample rate of 512, support display set to 1, and support locate set to 0. The locate() query was disabled to make the RLCSA index comparable with that of RLZ, but there was no option in the LZ-End implementation to disable the locate() query. Therefore, the LZ-End index may be larger than the indexes for COMRAD and RLCSA.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

**Results**

Table 7.1 contains the display() query times for each collection. Overall, for all test query lengths for all collections, the display() query of RLZ has the fastest symbol extraction times. Both GDC and RLZ are consistently faster than the remaining three algorithms. As discussed in Section 5.3.3, COMRAD, RLCSA and LZ-End must traverse complex data structures before retrieving symbols, and often the memory locality of these accesses is low, making retrieval slower.

The RLZ query tends to be two to three times faster than the GDC query. This is expected, since RLZ has direct access to the positions of factors and the symbols in the reference, while GDC has to decode the positions and then the symbols. The drawback of RLZ is that the size of the index tends to be larger compared to GDC as shown in Table 7.2. However, the difference is not significant, and the faster query performance of RLZ is an advantage. Compared to the COMRAD, RLCSA and LZ-End indexes, the RLZ and GDC indexes are smaller in size. However, note that the LZ-End index also implements search queries, hence the larger size is most likely due to the extra data structures required to implement these queries. Overall, in terms of the display() query, RLZ and GDC have the best performing indexes.

In terms of the index construction time, COMRAD, RLCSA and LZ-End were significantly slower than RLZ and GDC. For example, LZ-End, COMRAD, RLCSA, RLZ, and GDC have index construction times of 2023, 834, 749, 216 and 32 seconds, respectively for the Spara collection. The construction times for LZ-End, COMRAD and RLCSA are much higher for the remaining two collections, whereas both RLZ and GDC can construct these indexes in 200–300 seconds. Overall, GDC is able to construct an index faster than RLZ due to using a hash table for factorisation instead of a suffix array.
7.1. **THE DISPLAY() QUERY**

<table>
<thead>
<tr>
<th>Collection</th>
<th>Index</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sson</td>
<td>RLZ</td>
<td>0.232</td>
<td>0.031</td>
<td>0.011</td>
<td>0.008</td>
<td>0.008</td>
</tr>
<tr>
<td></td>
<td>GDC</td>
<td>0.651</td>
<td>0.091</td>
<td>0.027</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td></td>
<td>COMRAD</td>
<td>8.074</td>
<td>0.835</td>
<td>0.117</td>
<td>0.042</td>
<td>0.034</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>17.634</td>
<td>2.350</td>
<td>0.843</td>
<td>0.693</td>
<td>0.677</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.402</td>
<td>0.537</td>
<td>0.433</td>
<td>0.423</td>
<td>0.420</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ</td>
<td>0.232</td>
<td>0.032</td>
<td>0.011</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td></td>
<td>GDC</td>
<td>0.685</td>
<td>0.082</td>
<td>0.214</td>
<td>0.014</td>
<td>0.013</td>
</tr>
<tr>
<td></td>
<td>COMRAD</td>
<td>4.031</td>
<td>0.433</td>
<td>0.071</td>
<td>0.036</td>
<td>0.033</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>17.997</td>
<td>2.377</td>
<td>0.847</td>
<td>0.689</td>
<td>0.675</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.206</td>
<td>0.510</td>
<td>0.418</td>
<td>0.407</td>
<td>0.406</td>
</tr>
<tr>
<td>Athal</td>
<td>RLZ</td>
<td>0.225</td>
<td>0.030</td>
<td>0.011</td>
<td>0.009</td>
<td>0.008</td>
</tr>
<tr>
<td></td>
<td>GDC</td>
<td>0.447</td>
<td>0.059</td>
<td>0.018</td>
<td>0.012</td>
<td>0.013</td>
</tr>
<tr>
<td></td>
<td>COMRAD</td>
<td>1.082</td>
<td>0.146</td>
<td>0.053</td>
<td>0.043</td>
<td>0.042</td>
</tr>
<tr>
<td></td>
<td>RLCSA</td>
<td>23.238</td>
<td>3.099</td>
<td>1.063</td>
<td>0.861</td>
<td>0.844</td>
</tr>
<tr>
<td></td>
<td>LZ-End</td>
<td>1.677</td>
<td>0.798</td>
<td>0.681</td>
<td>0.662</td>
<td>0.662</td>
</tr>
</tbody>
</table>

Table 7.1: The display() query times in μsec/base for RLZ, GDC, COMRAD, RLCSA and LZ-End for varying query lengths on the Sson, Spara and Athal collections.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sson</th>
<th>Spara</th>
<th>Athal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLZ</td>
<td>24.11</td>
<td>27.42</td>
<td>59.49</td>
</tr>
<tr>
<td>GDC</td>
<td>24.66</td>
<td>16.35</td>
<td>52.59</td>
</tr>
<tr>
<td>COMRAD</td>
<td>62.71</td>
<td>72.82</td>
<td>271.41</td>
</tr>
<tr>
<td>RLCSA</td>
<td>54.69</td>
<td>47.35</td>
<td>175.92</td>
</tr>
<tr>
<td>LZ-End</td>
<td>54.71</td>
<td>55.58</td>
<td>262.29</td>
</tr>
</tbody>
</table>

Table 7.2: The sizes in Mbytes of the indexes constructed for each of the collections, Sson, Spara and Athal, by each of the algorithms, RLZ, GDC, COMRAD, RLCSA and LZ-End, to support the display() query. Original collection sizes are 966.46, 429.27 and 506.67 Mbases, for Sson, Spara and Athal, respectively.

7.1.6 **Discussion**

The above results showed that RLZ has the best display() query performance compared to several other compressors that also implement this query. To support this query, the compressed output must be altered, where the reference is stored uncompressed, and instead of storing factor lengths, a compressed bit vector that indicates the factor start positions is stored. The result of these changes is that the compressed output is larger than the standard RLZ output. The same is true for the GDC algorithm [Deorowicz and Grabowski, 2011]. We discuss two improvements to reduce the space usage, which come at a cost to the query performance.

The first improvement is to store the reference as an FM-Index, which was an option discussed in Section 6.3. Instead of using $n_1 \log |\sigma'|$ bits, the FM-Index representation of $T_1$ can be stored using $n_1 H_k(T_1)$ bits. During the display() query, substrings need to be efficiently retrieved from the reference. Using the FM-Index representa-
tion of the reference, retrieving an \( \ell \) length substring requires \( O(n_1 \log^{1+\epsilon} + \ell \log |\sigma'|) \) time. Therefore, the overall time complexity of a single display() query is \( O(\log \frac{N}{Z} + n_1 \log^{1+\epsilon} + \ell \log |\sigma'|) \), including the time taken to execute the rank_1 query. This is clearly greater than the \( O(\log \frac{N}{Z} + \ell) \) time required by the current implementation, which was also emphasised by the time taken to decompress the Spara collection in practice, in Section 6.3. Also no space savings were made in practice when an FM-Index representation was stored instead of a byte-packed reference. Therefore, we did not include this improvement in our implementation.

Another option is to store the symbols in the reference as Huffman codes, which is an alternative attempted by Deorowicz and Grabowski [2011]. As shown by their experiments, the query performance with this alternative representation is slower. To retrieve a substring from the reference, the blocks containing the substring needs to be decompressed rather than just the substring of interest. Overall, it is important to have fast access to symbols in the reference for efficient query performance.

The second improvement we propose is to the storage of the positions array. Currently the factor positions are stored in an array using \( \log n_1 \) bits per entry. However, factors do not necessarily begin at all positions in the range \([0, n_1]\). Therefore, it is unnecessary to use \( \log n_1 \) bits per position entry, and could in practice use less space by mapping the positions to consecutive integers. As an example, for a reference of length \( 10 \), if only the positions \([1, 3, 5, 6, 10]\) are used, then the mapping \( \{1 \rightarrow 0, 3 \rightarrow 1, 5 \rightarrow 2, 6 \rightarrow 3, 10 \rightarrow 4\} \) can be used to store the positions with \( \lceil \log(4 + 1) \rceil \) bits per entry instead of \( \lceil \log(10 + 1) \rceil \) bits. However, a compressed bit vector \( S \) of length \( n_1 \) is required, with a 1-bit at each position where a factor position starts, to implement the mapping and the inverse mapping. Let the mapped positions array be \( P' \), where \( P'[i] = \text{rank}_1(S, i) - 1 \) for \( 0 \leq i < Z \). The inverse mapping can then be obtained as \( P'[i] = \text{select}_1(S, P'[i] + 1) \). We implemented the \( S \) bit vector using the compressed bit vector by Raman et al. [2002], which occupies \( n_1 H_0(S) + o(n_1) \) bits and can perform the rank_1 and select_1 queries in \( O(1) \) time. For the Spara collection, the \( S \) bit vector required approximately 1 Mbyte and the \( P \) array is reduced in size by approximately 2 Mbytes. Therefore, the overall space saving is approximately 1 Mbyte, where the original index was 27.42 Mbytes, and the index with \( P' \) and \( S \) is 26.41 Mbytes. In terms of query performance, the speed is slower by around 0.07 microseconds/base for the queries of length 10, and by less than 0.01 microseconds/base for the longer queries. Therefore, the impact on the query performance is minimal. The other benefit of the \( S \) compressed bit vector is that it is required to implement the search queries, as discussed in Section 7.2.

The index we constructed used the standard RLZ algorithm described in Section 6.1 and does not use the features of the improved RLZ algorithm described in Section 6.2. Recall that the three improvements in the optimised RLZ algorithm are the non-greedy Lempel-Ziv parsing, efficient short factor encoding, and efficiently encoding the position components of factors that form an alignment (LISS factors).
Using non-greedy Lempel-Ziv parsing has the effect of increasing the total number of factors, therefore, constructing an \texttt{RLZ} index with non-greedy parsing increases the size of the index, unless alternatives are used to store short factors in a different manner. We experimented with storing the substrings associated with short factors directly using \( \log |\sigma'| \) bpb instead of storing a position value. An extra compressed bit vector of length \( Z \) is required to distinguish between short factors and standard factors. Overall, our experiments showed that the extra compressed bit vectors required to allow constant time access to factors and symbols require almost as much space as the space saved by non-greedy factorisation and explicitly storing the short factors. Similarly, when LISS factors are considered, a compressed bit vector of length \( Z \) is required to identify LISS factors, and data structures to store the LISS factor information. The space usage of this additional information was greater than the space saving of using LISS factors.

Two of the most important queries supported by self-indexes are the \texttt{count()} and \texttt{locate()} queries, which permit counting of and location of the occurrences of a given substring in the compressed collection. Next we discuss the modifications required to implement these two queries with some limitations for an \texttt{RLZ}-compressed collection.

## 7.2 Search: \texttt{count()} and \texttt{locate()} queries

The \texttt{count()} and \texttt{locate()} queries provide two of the key functionalities of a self-index: the ability to search for the occurrences of a given substring. In this section, we discuss the additional data structures required to extend the \texttt{RLZ} index presented in the previous section to implement the search queries, and also discuss the query algorithms. First we outline the proposed search algorithm to show why the additional data structures are necessary. Then we discuss the new data structures, followed by the detailed algorithm for implementing the search functionality. Note that our \texttt{RLZ} self-index implementation has a limitation, since we assume that a pattern being searched for will not cross more than two factors.

Let \( T \) be the original collection of \( r \) DNA sequences, and let \( T_c \) be the compressed \texttt{RLZ} index of \( T \). The basic search query is, given a substring \( X \), return the number of occurrences of the substring \( X \) (\texttt{count()}), or the positions of the occurrences of the substring \( X \) (\texttt{locate()} in \( T \), using the compressed index \( T_c \). Recall from Definition 6 that \texttt{RLZ} stores each sequence \( T_i \) in \( T \) as a set of \( z_i \) factors, where a factor represents an occurrence of a substring shared between \( T_i \) and the reference string \( T_1 \).

There are three types of occurrences of a substring \( X \) in \( T_c \) as illustrated in Figure 7.1. The first type is the occurrences of \( X \) in \( T_1 \). The second type is the occurrences that are completely contained within a factor. The third type is the ones that occur across two or more factors. We define the first and second type as \textit{primary occurrences} and the third type as \textit{secondary occurrences}. Note that we have switched the definitions of primary and secondary occurrences from Section 3.4.3.
Since the sequences $T_2 \ldots T_r$ are represented in terms of the reference $T_1$, the occurrences of $X$ or substrings of $X$ must first be found in the reference. Then the positions at which these occurrences are found in the reference can be used to search for occurrences in the remaining sequences. For primary occurrences, the reference $T_1$ is searched for $X$, to obtain the positions of occurrences $\text{Occ}_{T_1}^X$. Then for each position $p \in \text{Occ}_{T_1}^X$, the set of factors $Z$ in $T_c$ need to be searched to find all factors that completely contain the interval $[p, p + |X|)$. The number of factors containing the interval $[p, p + |X|)$ are the number of primary occurrences of substring $X$ in $T$.

To detect secondary occurrences, $X$ is split into two substrings $X_1, X_2$, and pairs of factors, where $X_1$ is a suffix of the first factor and $X_2$ is a prefix of the second factor is searched for. The substring must be split in all possible ways to find all such factor pairs. Here we assume that substrings being searched for span across at most two factors. This is a limitation of our algorithm and the search for substrings spanning across more than two factors is left as future work.

To find factor pairs where one ends with $X_1$ and the other starts with $X_2$, first we search $T_1$ to find the positions of occurrence $\text{Occ}_{T_1}^{X_1}$ of $X_1$. For each position $p_j = \text{Occ}_{T_1}^{X_1}[j]$, we search the factors in $T_c$ to find all factors $F_{p_j + |X_1|}$ that end at position $p_j + |X_1|$. Then the $(k+1)$th factor, where $k \in F_{p_j + |X_1|}$, is examined to check if it begins with $X_2$, and if so the $k$th and $(k+1)$th factor pair contains substring $X$ and the pair is counted as an occurrence. The process continues for all ways of splitting the substring $X$ into a prefix and a suffix.

This description of the substring searching process is an overview to describe the new data structures added to the RLZ index to enable search queries. The detailed search algorithm is presented in Section 7.2.2. Next we discuss the modifications required to the compressed output of the RLZ algorithm to support the search queries.

### 7.2.1 Modifications to the RLZ index

Recall from Section 7.1.1 that the RLZ index stores the reference string $T_1$, the position components of the factors $P$, the factor start positions in each sequence $L$, and the cumulative sequence lengths $B$. These data structures contain sufficient information to perform $display()$ queries. The search algorithm contains two queries that cannot be efficiently supported by the current data structures. One is the ability to obtain all occurrences of a substring $X$ from $T_1$. The other is the ability
to query the set of factors to determine which factors contain a particular interval in the reference, and which factors start or end at a particular position of the reference. Using the current data structures, a linear search through $T_1$ and a linear search through the factors is required to perform those queries, respectively. To lower the cost of searching $T_1$ for $X$, the suffix array $SA_{T_1}$ is included in the index. The substring can then be searched in $O(|X| \log n_1)$ time. Recall from Section 6.1.2 that $SA_{T_1}$ is used during the RLZ parsing step. Unfortunately, the suffix array requires $n_1 \log n_1$ bits of storage and this is a significant cost. Therefore, we consider storing the compressed suffix array, and the resulting effect on the index size and the query performance is experimentally analysed in Section 7.2.4.

For the remainder of the section, we discuss RLZ factors in terms of intervals, since each factor is a position and length pair $(p, \ell)$, which can be considered as an interval $[p, p + \ell)$. To search for factors that start or end at a given position, the factors can be sorted in increasing start position order and increasing end position order. However, maintaining two separate orderings is expensive. Also, neither start nor end position ordering of factors assist in searching for factors containing a given interval. If the factors are ordered according to the start position, then an interval $[a, b)$ can be contained within any of the factors with a start position $p \leq a$. Similarly, if the factors are ordered according to the end position, then an interval $[a, b)$ can be contained within any of the factors with an end position $p \leq b$. Factors can be ordered using both the start and end positions, but a strict ordering that permits ruling out factors that end before $a$ and start at or after $b$ is not possible due to the existence of factors that are completely contained within other factors.

This problem was also encountered by Kärkkäinen and Sutinen [1998] during their construction of a Lempel-Ziv index for $q$-gram searching. They proposed a data structure named nesting level lists, which is a multi-level list, where the first level contains intervals that have a $\leq$ ordering (for two intervals $[a, b)$ and $[a', b')$, $a \leq a'$ and $b \leq b'$). The second level contains intervals, all of which are contained completely within at least one interval in the first level (for two intervals $[a, b)$ and $[a', b')$, $[a', b')$ is in the second level if $a' \geq a$ and $b' < b$ or $a' > a$ and $b' \leq b$, where $[a, b)$ is in the first level). The intervals in the second level also have a $\leq$ ordering. More generally, all intervals within a level $k + 1$ is contained completely within at least one interval of level $k$, and the intervals in each level $k$ has a $\leq$ ordering. Let $NLL$ be a nesting level list with $K$ levels. At each level $k$ of $NLL$, given an interval $[c, d)$, if the level has intervals that contain the given interval, it is possible to calculate left and right boundaries $lb_k$ and $rb_k$ such that for all $j \in [lb_k, \ldots rb_k]$, $NLL[k][j]$ contains $[c, d)$. An example nesting level list is illustrated in Figure 7.2.

We adopt the nesting level list data structure for the RLZ index with slight modifications to support searching for intervals that contain a given interval. Let $NLL$ be the nesting level list for the factors in the RLZ-compressed collection $T_c$. The structure proposed by Kärkkäinen and Sutinen [1998] uses pointers between a
Level 3: [10,13)
Level 2: [7,14)
Level 1: [3,5) [6,15) [17,19)
Level 0: [1,5) [2,10) [5,15) [11,20)

Figure 7.2: An example of a nesting level list with four levels.

(a) NLL after inserting first interval
(b) NLL after inserting second interval
(c) NLL after inserting third interval

Figure 7.3: The nesting level list for the sorted intervals [1,5), [2,10), [3,5).

level \( k + 1 \) interval and the intervals in level \( k \) that contain this interval. As a result, once a higher-level interval is found that contains the interval of interest, then the pointers can be followed to find other intervals containing the interval of interest in constant time per occurrence. We do not store information about which interval contains which other interval to reduce the space usage. Instead we binary search each level of \( NLL \) to find intervals that contain the interval of interest. We store the \( NLL \) data structure as an array of length \( Z \), where all the levels are concatenated. An array \( NLL_{idx} \) of length \( K \), where \( K \) is the number of levels in the \( NLL \), is used to access each level in \( NLL \). The \( NLL \) itself is an index into the factors so that the original order of factors are preserved by the \( P \) array and \( L \) bit vector. The \( NLL \) data structure requires \( Z \log Z \) bits and \( NLL_{idx} \) requires \( K \log Z \) bits.

To build the \( NLL \) data structure, we sort the factors in increasing start position order, and, if factors have the same start position, then they are sorted on decreasing end position order. Let \( F \) be the set of sorted factors. Initially, \( NLL \) contains a single level, so \( K = 1 \). Each factor \( F[j] \), where \( 0 \leq j < Z \), is compared with the last entry in each level \( k \), where \( 0 \leq k < K \). Factor \( F[j] \) is appended to an existing level \( k \), if the last entry of that level does not contain \( F[j] \). If \( F[j] \) is contained within intervals in all \( K \) existing levels, then a new level is added, \( K \) is incremented, and \( F[j] \) is appended to the new level. Figure 7.3 illustrates the \( NLL \) construction process. The \( NLL_{idx} \) is constructed after the \( K \) levels of the \( NLL \) are concatenated.

To search for factors that start or end at a given position, we also binary search each level of the \( NLL \), since each level has a \( \leq \) ordering. Therefore, the \( NLL \) and \( SA_F \) structures can be used to answer all queries required by the search algorithms.

We also consider using two other compressed bit vectors to narrow the search space of the \( NLL \) when searching for factors that start or end at a certain position of the reference. As we discussed earlier in Section 7.1.6, factors do not start at all positions of the reference. Similarly, factors do not end at all positions of the reference. When searching for occurrences of \( X \) that split across two factors, we
either search \( NLL \) for factors that end at positions where \( X_1 \) ends in the reference, or factors that start at positions where \( X_2 \) starts in the reference. Searching \( SA_{T_1} \) for \( X_1 \) returns all occurrences of the substring in \( T_1 \), but factors will not end at some of those positions. Therefore, the cost of searching for certain end positions in \( NLL \) can be avoided if a bit vector is available that could indicate the positions at which factors end. Similarly, a bit vector that indicates the positions at which factors start will reduce the search space when searching for factors that start with the prefix \( X_2 \). In Section 7.1.6, we discussed the compressed bit vector \( S \) of length \( n_1 \) that indicates the positions at which factors start so that the positions in \( X \) can be mapped to consecutive integers. We use this bit vector \( S \) to reduce the searches made on the \( NLL \) for start positions. Similarly, we also use a bit vector \( E \) of length \( n_1 \) that contains a 1-bit at positions of the reference where factors end, to reduce the searches made on the \( NLL \). We use the RRR representation of Raman et al. [2002] for both \( S \) and \( E \) compressed bit vectors, which require \( n_1 H_0(S) + o(n_1) \) and \( n_1 H_0(E) + o(n_1) \) bits, respectively.

The final data structure we add is another compressed bit vector of length \( Z \). Since the factors for each sequence are concatenated together, a bit vector with a 1-bit that indicates the position of the first factor of each sequence is necessary to determine the sequence that an occurrence of a substring belongs to in constant time. This bit vector identified as \( FB \) is stored using the sdarray representation [Okanohara and Sadakane, 2007] in \( r \log \frac{Z}{r} + O(r) \) bits.

In summary, the data structures added to the compressed \( RLZ \) index are: the suffix array of the reference \( SA_{T_1} \), the searchable factors \( NLL \), the index into the levels of the searchable factors \( NLL_{idx} \), the compressed bit vectors indicating the positions in the reference at which factors start and end, \( S \) and \( E \), and the compressed bit vector indicating sequence boundaries in the concatenated factors, \( FB \).

7.2.2 The search algorithm

We now discuss the \( RLZ \) search algorithm in detail. The algorithms for \( count() \) and \( locate() \) queries are almost identical, except that the \( locate() \) query must also report the sequence and position of the sequence at which an occurrence was found. We present the algorithm for the \( count() \) query, and later discuss how to obtain the sequence and position of occurrence.

\( count() \)

Let the compressed \( RLZ \) index of \( r \) sequences be \( T_c \), and let sequence \( T_1 \) be the reference. Sequences \( T_2 \ldots T_r \) are represented by \( Z \) factors, and the compressed index \( T_c \) contains the data structures that allow factors to be searched efficiently. Let the search query be \( count(X) \), where substring \( X \) has a length of \(|X|\). We also assume that all occurrences of the substring span across at most two factors.
Algorithm 11 RLZ\_count\_primary\_occ\((X, T_1, SAT_1, NLL, K)\) takes a substring \(X\), a reference string \(T_1\) and its suffix array \(SAT_1\), a nesting level list containing the sorted factors \(NLL\) and the number of levels \(K\) as arguments, and returns the number of primary occurrences of the substring \(X\).

1. \(\text{Occ}_{X}^{T_1} \leftarrow \text{search\_ref}(T_1, SAT_1, X)\) \{Step 1: Primary occurrences in \(T_1\}\)
2. \(\text{occ} \leftarrow |\text{Occ}_{X}^{T_1}|\)
3. \(\text{for all } p \text{ in } \text{Occ}_{X}^{T_1} \text{ do} \) \{Step 2: Primary occurrences in compressed sequences\}
4. \(\text{ for all } k \text{ in } [0 \ldots K] \text{ do} \)
5. \((lb_{NLL_k}, rb_{NLL_k}) \leftarrow \text{bsearch\_interval\_null}(NLL, k, p, p + |X|)\)
6. \(\text{occ} \leftarrow \text{occ} + (rb_{NLL_k} - lb_{NLL_k})\)
7. \(\text{end for}\)
8. \(\text{end for}\)
9. \(\text{return } \text{occ}\)

Earlier we defined two types of substring occurrences. Primary occurrences are occurrences found in the reference and occurrences contained within single factors in \(T_e\). Secondary occurrences span across two factors. Initially, the primary occurrences of \(X\) in the reference are found by binary searching \(SAT_1\). If \(X\) does not occur in \(T_1\), then no primary occurrences exist and the search continues with secondary occurrences. If \(X\) occurs in \(T_1\), then the positions of occurrence of \(X\) in \(T_1\) are \(\text{Occ}_{X}^{T_1} = SAT_1[lb_{SA} \ldots rb_{SA}]\), where \(lb_{SA}\) and \(rb_{SA}\) are the suffix array boundaries for suffixes prefixed with \(X\). Then for each position \(p \in \text{Occ}_{X}^{T_1}\), the \(NLL\) structure is searched to find intervals containing the interval \([p, p + |X|]\), which corresponds to an occurrence of \(X\) in \(T_1\). If the \(NLL\) contains \(K\) levels, then each level is binary searched to find intervals containing \([p, p + |X|]\). For each level \(k\), where \(k \in 0 \ldots K - 1\), the binary search returns a lower and upper bound \(lb_{NLL_k}\) and \(rb_{NLL_k}\) such that a factor \((P[j], select_1(L, j + 2) - select_1(L, j + 1))\) contains the substring \(X\), where \(j \in NLL[lb_{NLL_k} \ldots rb_{NLL_k}]\). Therefore, the total primary occurrences are \(\text{Occ}_{X}^{T_1} + \sum_{g}^{\text{Occ}_{X}^{T_1}} \sum_{k=0}^{K}(rb_{NLL_k} - lb_{NLL_k})\). The first term is the number of occurrences in the reference. The second term is the number of occurrences in the remaining sequences, which are searched for using each of the occurrences in the reference.

Primary occurrence search is illustrated in Algorithm 11.

To search for secondary occurrences, the substring \(X\) is split in all \(|X| - 1\) ways to a prefix \(X_1\) and a suffix \(X_2\). For simplicity, we present the secondary occurrence search algorithm in terms of first finding factors that have \(X_1\) as a suffix, then narrowing the set of factors to those that have the next factor prefixed with \(X_2\). First the string \(T_1\) is searched for occurrences of \(X_1\). Let these occurrences be \(\text{Occ}_{X_1}^{T_1}\). Then, for each position \(p \in \text{Occ}_{X_1}^{T_1}\), if \(E[p + |X_1|]\) is true, each level of the \(NLL\) is binary searched to find factors ending at position \(p + |X_1|\). For each \(j \in NLL[lb_{NLL_k} \ldots rb_{NLL_k}]\), at most \(|X_2|\) symbols of the substring represented by the \((j + 1)\)th factor is checked to see if the \(j\)th and \((j + 1)\)th factors form a factor pair that has an occurrence of \(X\) across it. If factors \(j\) and \(j+1\) form such a pair, then
Algorithm 12 \textit{RLZ\_count\_secondary\_occ}(X, T_1, SA_{T_1}, NLL, K, P, L) takes a substring \(X\), a reference string \(T_1\) and its suffix array \(SA_{T_1}\), a nesting level list containing the sorted factors \(NLL\) and the number of levels \(K\), the list of factor positions \(P\), and the factor start positions \(L\) as arguments, and returns the number of secondary occurrences of the substring \(X\).

1: \(\text{occ} \leftarrow 0\)
2: for all \(x\) in \([1 \ldots |X|]\) do
3: \(X_1 \leftarrow X[0 \ldots x], X_2 \leftarrow X[x \ldots |X|]\)
4: \(\text{Occ}_{T_1}^{X_2} \leftarrow \text{search\_ref}(T_1, SA_{T_1}, X_2)\) \{Step 1: Search for \(X_2\) in the reference\}
5: for all \(p\) in \(\text{Occ}_{T_1}^{X_2}\) do \{Step 2: Search for factors starting with \(X_2\)\}
6: \(\text{continue}\) if no factors start at \(p\)
7: \(\text{Occ}_{NLL}^{X_2} \leftarrow \text{search\_start\_nll}(NLL, K, p)\) \{Get all factors starting at \(p\}\}
8: for all \(j\) in \(\text{Occ}_{NLL}^{X_2}\) do \{Step 3: Check if next factor starts with \(X_2\)\}
9: \(\text{pos} \leftarrow P[j - 1], \ell \leftarrow \text{select}_1(L, j + 1) - \text{select}_1(L, j)\)
10: if \(X_1 == T_1[\text{pos} + \ell - |X_1| \ldots \text{pos} + \ell]\) then
11: \(\text{occ} \leftarrow \text{occ} + 1\)
12: end if
13: end for
14: end for
15: end for
16: return \(\text{occ}\)

it is counted as an occurrence. The process continues for all the prefix and suffix pairs. The secondary occurrences are counted as each occurrence is encountered. The secondary occurrence search is illustrated in Algorithm 12.

In practice, we implement the search algorithm differently depending on the lengths of the prefix and suffix components. If the prefix \(X_1\) is short, a large fraction of the factors in the collection end with this prefix. Then in step 3 of Algorithm 12, the number of factors to check if they begin with \(X_2\) is high, making the algorithm slow in practice. To reduce the possibility of this event, Algorithm 12 is only conducted for the cases where \(|X_1| \geq |X_2|\). Then for the cases where \(|X_1| < |X_2|\), in a manner very similar to that of Algorithm 12, factors starting with \(X_2\) are searched for in the \(NLL\), then the resulting factors are narrowed to those that have a previous factor ending with \(X_1\) to calculate the occurrences.

\texttt{locate()}

The algorithm for \texttt{locate()} is the same as that for \texttt{count()}, except when an occurrence is detected, it is necessary to determine the sequence and position at which the occurrence originates. For the primary occurrences in the reference, this is trivial, since the positions of occurrence can be extracted directly from the suffix array. For the primary occurrences in the remaining sequences, binary searching each level of the \(NLL\) provides the identifiers for factors that contain the substring. The factor identifiers can then be used to determine the sequence and position of each
occurrence. As an example, if the $j$th factor in the collection contains the substring of interest, a $i = \text{rank}_1(FB, j)$ query is invoked to establish that the $j$th factor belongs to sequence $i$. Then $\text{facpos} = \text{select}_1(L, j + 1) - B[i]$ is the position in the $i$th sequence at which the $j$th factor begins. Therefore, $\text{facpos} + \text{offset}$ is the position at which the factor occurs in the $i$th sequence, where $\text{offset} = p - P[j]$, and $p$ is the position in the reference of the occurrence that corresponds to the occurrence in factor $j$. For secondary occurrences, assuming an occurrence spans across the $j$th and $(j + 1)$th factor, the position of occurrence can be calculated from the $j$th factor in the same manner as for primary occurrences.

### 7.2.3 Time and space complexity

The space complexity of the additional data structures required to support the search query was discussed in Section 7.2.1. A summary of the space usage of the data structures in the RLZ index is as follows.

- Reference string ($T_1$): $n_1 \log |\sigma'|$ bits
- Suffix array of the reference ($SA_{T_1}$): $n_1 \log n_1$ bits
- Array of factor positions ($P$): $Z \log n_1$ bits
- Factor start positions in the sequences ($L$): $Z \log \frac{N}{Z} + O(Z)$ bits
- Cumulative sequence lengths ($B$): $r \log N$ bits
- Sorted factor index ($NLL$): $Z \log Z$ bits
- Index into the levels of the $NLL$ ($NLL_{idx}$): $K \log Z$ bits
- Positions in the reference at which factors start ($S$): $n_1 H_0(S) + o(n_1)$ bits
- Positions in the reference at which factors end ($E$): $n_1 H_0(E) + o(n_1)$ bits
- Sequence boundaries for concatenated factors ($FB$): $r \log \frac{Z}{r} + O(r)$ bits

The index size can be reduced significantly by using a compressed suffix array. The space for factor positions can also be reduced as described in Section 7.1.6, by mapping the positions to consecutive integers. The savings from these improvements and their impact on query performance is experimentally verified in Section 7.2.4.

During index construction, all the factors need to be stored in memory to build the data structures. The only additional major cost of constructing the index is in constructing the $NLL$ data structure. As stated earlier, the factors are sorted in increasing start position and decreasing end position order, which requires $O(Z \log Z)$ time. Additional space of $Z \log Z$ bits is also required, since we sort an index of the factors rather than the factors themselves. Once the factors are sorted, the $NLL$
can be constructed in $O(Z)$ time. For the remaining data structures $S$, $E$ and $FB$, bit vectors are maintained during the RLZ factorisation, which requires $Z$, $Z$ and $r$ bits, respectively. Once the factorisation is complete, the compressed bit vectors are calculated. If a compressed suffix array is used, we specifically construct this data structure for the index, since we do not use the compressed variant during compression as it makes the compression step significantly slower.

The time complexity of the $\text{count()}$ query can be evaluated by examining each step of the query algorithm. First, the primary occurrences in the reference are determined with a binary search of the suffix array, which requires $O(\log n)$ time. Let $occ^{T_1}_X$ be the number of primary occurrences of substring $X$ in the reference. Then each of the $K$ levels of the $NLL$ data structure are binary searched for each primary occurrence in the reference. In the worst case, the $NLL$ structure contains a single factor in each level, resulting in $K = Z$ levels. Therefore, $O(Z)$ time is required to determine the factors that contain a given interval. Therefore, the cost of searching for primary occurrences is $O(occ^{T_1}_X + |X| \log n_1)$. In practice, the factors are distributed among a few levels with the lower levels containing most of the factors, and the higher levels containing fewer factors. Therefore, the cost of primary occurrence searching is significantly less than $O(Z)$.

When searching for secondary occurrences, the substring $X$ is divided into a prefix and a suffix component in $|X| - 1$ possible ways. Recall from the previous section that, when $|X_1| < |X_2|$, the suffix $X_2$ is searched for in $T_1$, whereas, when $|X_1| \geq |X_2|$, then prefix $X_1$ is searched for in $T_1$. The search algorithm is the same regardless of which partition is searched for in the $NLL$, therefore we generalise the time complexity calculation. Let $X'$ be the partition being searched for in $T_1$, and let $X''$ be the other partition. Searching for $X'$ in $SA_{T_1}$ requires $O(|X'| \log n_1)$ time. Let $occ^{T_1}_{X'}$ be the number of occurrences of substring $X'$ in the reference. Then each of the $K$ levels of the $NLL$ data structure are binary searched for each occurrence of $X'$ in the reference. This requires $O(occ^{T_1}_{X'} Z)$ time in the worst case. In the worst case, all factors may be prefixed or suffixed by $X'$, depending on whether $X'$ is the prefix partition or the suffix partition. Then for each such factor, the suffix of the substring associated with the previous factor, or the prefix of the substring associated with the next factor needs to be compared with substring $X''$, which requires $O(|X''| Z)$ time in the worst case. Overall, searching for a single partition requires $O(|X'| \log n_1 + occ^{T_1}_{X'} Z |X''|)$ time. To calculate the cost for all partitions, note that the partition $X'$ ranges in lengths between $|X|/2$ and $|X| - 1$, while partition $X''$ ranges in lengths between 1 and $|X|/2 - 1$. Therefore, the sum of the $|X'|$ lengths and the $|X''|$ lengths are proportional to $|X|^2$. Therefore, after considering all possible partitions, the worst case time complexity of searching for secondary occurrences is $O(|X|^2 (\log n_1 + occ^{T_1}_{X} Z))$.

Let $occ^{T_1}_p$ be the number of primary occurrences in the sequences $T_2 \ldots T_r$, and $occ^{T_1}_p$ is the primary occurrences in the reference. Also let $occ^{T_1}_s$ be the secondary
occurrences in the sequences $T_2 \ldots T_r$. Although $locate()$ and $count()$ mostly follow the same algorithm, the $locate()$ query also has to calculate the positions at which the occurrences are found. As stated earlier, to find the sequence at which a given occurrence originates at requires a $rank_1$ query on the $FB$ compressed bit vector. The $FB$ bit vector is stored in $sdarray$ format, hence requires $O(\log \frac{Z}{r})$ time for a $rank_1$ query. Finding the position of occurrence in the sequence takes constant time. Therefore, an additional $O((occ_{T_p}^{T_1} + occ_{T_s}^{T_1}) \log \frac{Z}{r})$ time is required by the $locate()$ query to determine the positions of occurrence.

Overall, for the $count()$ query, the worst case time complexity is $O(|X|^2 \log n_1 + (occ_{X}^{T_1} + |X|^2 occ_{X}^{T_1})Z)$, where $occ_{X}^{T_1}$ here refers to the total number of occurrences of all the $X'$ partitions in $T_1$. The worst case time complexity for the $locate()$ query is $O(|X|^2 \log n_1 + (occ_{X}^{T_1} + |X|^2 occ_{X}^{T_1})Z + (occ_{T_p}^{T_1} + occ_{T_s}^{T_1}) \log \frac{Z}{r})$. Note that the time complexity for the search queries depends on the number of factors in the collection rather than the number of occurrences of a given substring. It would be ideal to construct an index where the search complexity depends on the number of occurrences. However, it is unclear whether this is possible for an $LZ77$ index due to the need to find occurrences that cross factor boundaries. Next we experimentally evaluate the performance of the $RLZ$ self-index.

## 7.2.4 Experimental evaluation

We now experimentally evaluate the query performance of four versions of the $RLZ$ self-index to determine the time and space trade-offs of using various data structures. We also compare the query performance, index sizes and the index construction times to the existing self-indexes $LZ$-End and $RLCSA$. The experimental data and environment are discussed prior to the results.

### Test data and environment

We used the same test collections and environment used to evaluate the $display()$ query performance of $RLZ$ in Section 7.1.5. We selected the highly repetitive $Sson$, moderately repetitive $Spara$ and less repetitive $Athal$ collections from DATASET-REP, as the test collections. For each collection, we constructed four $RLZ$ self-indexes, all of which has various space time trade-offs. The first variant named $RLZ$-basic, is the basic index, which uses the original positions array $P$, and the uncompressed suffix array $SA_{T_1}$. This is likely to be the most space-consuming variant but should provide faster query performance compared to the remaining variants. The second variant named $RLZ$-$P'$, uses the more compact array of positions $P'$, described in Section 7.1.6, which contains the factor positions mapped to consecutive integers. This variant will use slightly less space than the first variant, but the query performance will be slightly slower, since upon each factor position access, it needs to be mapped to its original value. The third variant named $RLZ$-$CSA$, uses the compressed
7.2. \textbf{SEARCH: COUNT() AND LOCATE() QUERIES}

suffix array implementation in the \texttt{libcds} library by Sadakane [2000, 2002, 2003] instead of the uncompressed suffix array. The space use of the index will improve significantly with this variant, but the query performance will decrease compared to \texttt{RLZ-basic}, due to the overhead of accessing the compressed data structure. The final variant named \texttt{RLZ-P' CSA}, uses both the compact positions array \(P'\) and the compressed suffix array. This variant will also reduce the space use of the index, but the query performance is likely to be slower than the other three variants.

We randomly generated 100 substring queries with a query length of 10 each, ensuring that none of the occurrences spanned more than two factors. The query time was measured using the \texttt{gettimeofday()} function, and does not include the time taken to output the query result. Over the 100 queries, the time taken to answer each query and the number of occurrences were accumulated, then the total time was divided by the total occurrences to report the average number of micro seconds per occurrence. The results reported are the fastest of five runs.

We also compare the \texttt{RLZ} self-index to the \texttt{RLCSA} and \texttt{LZ-End} self-index implementations. \texttt{RLCSA} [Mäkinen et al., 2009; Mäkinen et al., 2010] and \texttt{LZ-End} [Kreft and Navarro, 2010, 2011] were introduced in Sections 3.4.2 and 3.4.3, respectively. We built an index for each of the three collections with \texttt{RLCSA} and \texttt{LZ-End}. All the compressors except \texttt{LZ-End} can build an index for multi-sequence collections. For \texttt{LZ-End}, the individual sequences were concatenated to create a single sequence. The \texttt{RLCSA} parameters used were, a block size of 32, sample rate of 512, support display set to 1, and support locate set to 1. \texttt{LZ-End} had no parameters that could be set.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

\textbf{Results}

Table 7.3 shows the query times in microseconds per occurrence for both the \texttt{count()} and \texttt{locate()} queries. Overall, all the \texttt{RLZ} self-index variants are slower than \texttt{RLCSA} and \texttt{LZ-End}. For the \texttt{count()} query, \texttt{RLCSA} is significantly faster, while for the \texttt{locate()} query \texttt{RLCSA} is slower than \texttt{LZ-End}. This is expected as the \texttt{RLCSA locate()} query requires access to the suffix array samples, while the \texttt{count()} query is answered simply by binary searching the suffix array. The performance of \texttt{LZ-End} is the same for both the \texttt{count()} and \texttt{locate()} queries. Like \texttt{RLZ}, the \texttt{LZ-End count()} and \texttt{locate()} algorithms are similar except for the need to access the locations of the occurrences, which does not add extra complexity.

The best performing \texttt{RLZ} self-index is \texttt{RLZ-basic} and the query performance is the same for both \texttt{count()} and \texttt{locate()}. However, \texttt{RLZ-basic} is 5–6 times slower than \texttt{LZ-End}. Moreover, reducing the space use of the \texttt{RLZ} self-index results in significant reductions in the query performance. In the case of \texttt{RLZ-P'}, the query performance reduced by around a factor of four, simply due to the \texttt{select1()} queries required to
CHAPTER 7. THE RLZ SELF-INDEX

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sson count() locate()</th>
<th>Spara count() locate()</th>
<th>Athal count() locate()</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLZ-basic</td>
<td>269.92</td>
<td>269.21</td>
<td>282.97</td>
</tr>
<tr>
<td>RLZ-P'</td>
<td>922.27</td>
<td>919.12</td>
<td>961.51</td>
</tr>
<tr>
<td>RLZ-CSA</td>
<td>570.65</td>
<td>571.73</td>
<td>1,064.89</td>
</tr>
<tr>
<td>RLZ-P'-CSA</td>
<td>1,250.62</td>
<td>1,260.17</td>
<td>1,815.54</td>
</tr>
<tr>
<td>RLCSA</td>
<td>0.02</td>
<td>45.62</td>
<td>0.04</td>
</tr>
<tr>
<td>LZ-End</td>
<td>36.14</td>
<td>35.31</td>
<td>43.09</td>
</tr>
</tbody>
</table>

Table 7.3: Query times measured in micro seconds per occurrence of a substring $X$, for the `count()` and `locate()` queries of the RLZ self-index variants, and RLCSA and LZ-End self-indexes.

remap position values. For RLZ-CSA, the query performance decreased due to the increased time required to access the elements in the compressed suffix array. The reduction in the query performance depended on the repetitiveness of the collection. For highly repetitive collections, there is a 2–4 fold decrease in query performance, while for the less repetitive Athal collection, the decrease in query performance is around 16 fold. This is most likely due to the large number of factors in less repetitive collections. For RLZ-P'-CSA, the reduction in query performance is even more significant, and like RLZ-CSA, the decrease in the performance depends on the repetitiveness of the collection. For the two most repetitive collections, the query performance decreased by 5–6 fold, while for the less repetitive Athal collection, the query performance decreased by around 20 fold.

Table 7.4 shows the index sizes for the RLZ self-index variants, and RLCSA and LZ-End self-indexes. RLZ-basic produces the largest self-indexes. For the highly repetitive Sson collection, the difference between the RLZ-basic index size and the RLCSA and LZ-End self-indexes is around 10 Mbytes. As the collections get less repetitive, the difference becomes more significant. The space saving from using RLZ-P' instead of RLZ-basic is not a significant enough improvement considering the resulting decrease in query performance. However, the space saving from using RLZ-CSA instead of RLZ-basic is significant, where approximately 20 Mbytes are saved for the two most repetitive collections, while the space use more than halves for the Athal collection. Once again, the space saving from using RLZ-P'-CSA instead of RLZ-CSA is also not worth while given the reduction in query performance. Therefore, out of the RLZ self-index variants, RLZ-CSA constitutes a better trade-off between space use and query performance. This is especially the case for the highly repetitive collections. Also, for the highly repetitive Sson collection, RLZ-CSA has a lower index size than RLCSA and LZ-End. However, the space saving from using the RLZ self-index may not be worth while considering the slow query performance. For the Spara and Athal collections, it is not worth while to use the RLZ self-indexes compared to the RLCSA or LZ-End indexes given the significantly slower query performance.
7.2. SEARCH: COUNT() AND LOCATE() QUERIES

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sson</th>
<th>Spara</th>
<th>Athal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLZ-basic</td>
<td>64.31</td>
<td>76.08</td>
<td>461.22</td>
</tr>
<tr>
<td>RLZ-P'</td>
<td>62.20</td>
<td>74.25</td>
<td>459.16</td>
</tr>
<tr>
<td>RLZ-CSA</td>
<td>47.38</td>
<td>56.28</td>
<td>210.55</td>
</tr>
<tr>
<td>RLZ-P'CSA</td>
<td>45.27</td>
<td>54.44</td>
<td>208.49</td>
</tr>
<tr>
<td>RLCBA</td>
<td>54.78</td>
<td>47.35</td>
<td>175.91</td>
</tr>
<tr>
<td>LZ-End</td>
<td>53.72</td>
<td>54.53</td>
<td>257.78</td>
</tr>
</tbody>
</table>

Table 7.4: Index sizes in Mbytes for the RLZ self-index variants, and the RLCBA and LZ-End indexes.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Sson</th>
<th>Spara</th>
<th>Athal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLZ-basic</td>
<td>204</td>
<td>187</td>
<td>250</td>
</tr>
<tr>
<td>RLZ-P'</td>
<td>207</td>
<td>188</td>
<td>261</td>
</tr>
<tr>
<td>RLZ-CSA</td>
<td>224</td>
<td>199</td>
<td>407</td>
</tr>
<tr>
<td>RLZ-P'CSA</td>
<td>206</td>
<td>196</td>
<td>389</td>
</tr>
<tr>
<td>RLCBA</td>
<td>1,803</td>
<td>711</td>
<td>999</td>
</tr>
<tr>
<td>LZ-End</td>
<td>4,802</td>
<td>2,023</td>
<td>2,812</td>
</tr>
</tbody>
</table>

Table 7.5: Index construction times in seconds for the RLZ self-index variants, and the RLCBA and LZ-End indexes.

The only advantage of the RLZ self-index is that it is significantly faster to construct than RLCBA and LZ-End self-indexes. The construction times are listed in Table 7.5. Overall, the RLZ index construction time is 4–9 times faster than RLCBA, while being 10–20 times faster than LZ-End. However, construction times for an index is not an important performance criteria in practice, since indexes only need to be constructed once. The performance criteria of importance is the query times, since once an index is constructed, thousands to millions of queries may be executed.

Overall, the current RLZ self-index is not practical for executing large sets of search queries due to the slow speed. Self-indexes such as RLCBA and LZ-End use only slightly more space than the most space-efficient RLZ self-index variant, but performs around 500 times faster. Therefore, further improvements need to be made to the RLZ self-index to improve the query performance such that it is practical for evaluating large sets of queries. Future work in this direction is discussed next.

7.2.5 Discussion

In this section we introduced a method to support the count() and locate() search queries on the RLZ compressed index to construct a full self-index with some limitations. The changes made to the RLZ index does not affect the existing display() query implementation. However, if the P' array is used instead of the P array, then as previously explained in Section 7.1.6, there is a slight reduction in the display() query speed due to the need to remap the factor position components.
The experimental analysis showed that search queries in all the RLZ self-index variants are significantly slower than the equivalent queries in the LZ-End and RLCSA indexes. The self-index variant using the $P'$ array and the compressed suffix array is the most space-efficient variant, even using less space than the LZ-End index, but it has the worst query performance. The variant that uses just the compressed suffix array provides the best trade-off between space usage and query performance. It may be possible to use an alternative compressed suffix array implementation that uses more space but improves the query speed. Currently, LZ-End is a better self-index in terms of the space use and the query performance. Significant improvements need to be made to the RLZ self-index to improve the query performance to be competitive with existing self-indexes.

When comparing the portion of the query time required to find primary and secondary occurrences, the primary occurrences, which constitute most of the occurrences, are found in less than half the time taken by LZ-End, even when the compressed suffix array is used. Therefore, most of the query time is spent in searching for secondary occurrences, hence the focus of the improvements should be in improving secondary occurrence detection. This is a difficult problem as pairs of adjacent factors need to be found that end with a prefix and begin with a suffix of the given substring. Constructing a data structure that retrieves such occurrences in time that is close to linear in the number of such occurrences is non-trivial. In the current solution, the time is linear in the number of occurrences of the prefix or suffix partition, and the number of such occurrences can be large for short prefixes or suffixes of the substring. One solution may be to have two orderings of the factors, one in forward lexicographic order of the substrings represented by factors, and the other in reverse lexicographic order of the substrings represented by factors. This is similar to the two patricia tries for forward and reverse phrases in LZ-End [Kreft and Navarro, 2011]. Unfortunately, in the case of the RLZ index, an extra $2Z \log Z$ bits of space are required to maintain the two orderings, which is a significant space overhead. Space efficient data structures suitable for secondary occurrence search needs to be explored in the future.

Improvements can also be made to primary occurrence detection. Currently, primary occurrences are detected by binary searching each level of the nesting level list $NLL$ for each occurrence of the substring in the reference. Instead, a two-dimensional grid containing a point at the row and column representing the start and end point of each factor, respectively, can be used to reduce the time complexity to be closer to $O(occ_p^c)$. The space usage of such a grid is unclear but compressed bit vectors can be used to represent the grid compactly while maintaining fast access. Note that only the top half of the diagonal between the top left corner and the bottom right corner of the grid needs to be stored, since the end positions of factors are strictly greater than the start positions of factors. It is also necessary to store pointers to the original factor list as the grid elements are not sufficient to identify the
sequence and position of occurrence for \textit{locate()} queries. Also more than one factor could have the same start and end location so pointers to the factors are required to distinguish between the multiple occurrences of the same factor. Therefore, as well as the space used for the grid, an additional $Z \log Z$ bits is required to store the pointers to the factors, making the space use greater than that of using the NLL data structure. Using the current algorithm for secondary occurrence search, the two-dimensional grid can also improve secondary occurrence search, since finding factors that start or end at a given position simply reduces down to accessing a row or column of the grid, respectively. Therefore, using the two-dimensional grid instead of the nesting level list can potentially improve the search speed.

Apart from the slow search speed, the major drawback is that the occurrences that span more than two factors are not detected. Such occurrences are possible if individual factors are short with respect to the length of the substring, especially if the substring crosses a factor that covers a mutation region. To search for substrings crossing more than two factors, another permutation of the factors is required, where the factors are ordered in increasing lexicographic order of the substring represented by the factor. If two factors represent the same substring, then the lexicographic ordering of the substrings represented by the subsequent factors is used to determine the ordering. Instead of storing the factors, an index into the factors is stored, which is a permutation of the numbers in the range $[0, Z)$. Now when a substring is split in all possible ways, the suffix of the substring is binary searched in the sorted factors, while the prefix is searched for in the NLL or the two-dimensional grid. Since the new data structure is ordered on substrings that cross multiple factors, searching for the substring suffix finds occurrences that cross factor boundaries. Unfortunately, an additional $Z \log Z$ bits are required to maintain this factor ordering.

The search algorithm also needs to be extended to find substrings containing $n$ symbols. If the RLZ algorithm is extended to handle all nucleotides in the extended alphabet, then substrings containing these extra nucleotides should also be searched for. The current difficulty of searching for such substrings is due to the different representation of factors containing $n$ symbols. If RLZ were to be extended to handle the extended DNA alphabet, then factors containing the extra nucleotides should be encoded in the same manner as the remaining factors to avoid special cases.

Another possibility is to extend the index to search for approximate matches. Approximate matches can be defined as matches where a substring $X$ is found in the text with at most $k$ errors, where the errors are mismatches, insertions or deletions. Especially in the context of DNA string matching, the ability to execute such a query is important, due to the presence of mutations. Approximate string searching using compressed suffix trees and suffix arrays has been explored [Crochemore and Tischler, 2010; Russo \textit{et al.}, 2009] and approximate matching in Lempel-Ziv indexes has also been studied [Russo \textit{et al.}, 2007]. It may be possible to adopt these solutions to the RLZ index to implement approximate substring searching.
As discussed above, several changes need to be made to the data structures of the RLZ self-index to improve the search speed of the algorithm. Special attention need to be paid to improve the search speed for substrings crossing more than one factor. The space and time trade-offs of the suggested improvements are unclear and these options will be explored in future work. Overall, the ability to perform search queries on the compressed collection is an important feature. The current implementation is sufficient but more practical self-indexes exist. Therefore, significant improvements are required to make the RLZ self-index competitive with existing indexes.

7.3 Chapter summary

In this chapter the RLZ compressor was transformed into a self-index that permits display(), count() and locate() queries to be executed on compressed sequence collections with certain limitations on the search queries. Supporting the display() query is the most natural extension to the RLZ compressor, since a small alteration to the factor storage is sufficient to efficiently implement the display() query on compressed collections. The experimental evaluation showed that the RLZ display() query is the fastest compared to existing indexes that also support the query. The drawback of supporting the display() query is that the compressed collection size is larger than when the query is unsupported, but the sizes are still significantly smaller than the original collection size.

Adding support for the count() and locate() queries were not as simple, and several additional data structures needed to be included in the compressed output to support these queries. The compressed collection sizes increased drastically as a result but for some of the variants, the index sizes were competitive with the two existing self index implementations, RLCSA and LZ-End. Unfortunately, the query performance of the RLZ self-indexes were significantly slower than the query performance of RLCSA and LZ-End, making it impractical for evaluating large sets of search queries. This is unfortunate as the count() and locate() queries are two of the most important queries supported by a self-index. The slow performance can be attributed to the cost of searching for occurrences that cross factor boundaries. To improve the speed of search queries, significant changes need to be made to the current implementation to include data structures that efficiently detect occurrences that cross factor boundaries. Also, a critical deficiency of the current algorithm is its inability to detect patterns occurring across more than two factors, which needs to be resolved for the index to be of practical use. The current implementation is sufficient to show that it is possible to extend the RLZ compressor to a full self-index.
Chapter 8

Reference construction\footnote{An earlier version of research presented in this chapter was published in Kuruppu et al. [2011b].}

In Chapter 6 we introduced a relative compressor called RLZ. Recall that relative compressors compress one or more sequences with respect to another sequence, which is known as the reference. For compression to be effective, a high level of similarity between the reference and the sequences being compressed is necessary. Typically, relative compression is suitable for compressing collections of homologous sequences. In Chapter 6 we showed that RLZ can produce excellent compression results for collections containing genomes from the same species. RLZ also compresses and decompresses fast, supports random access, and is memory-efficient.

The disadvantage of relative compression is that a collection being compressed is required to contain a sequence that can be used as a reference string. Ideally, this sequence will contain most of the repetitions present in the collection. If a collection contains some sequences that do not share much similarity with the reference, then these sequences will not compress well. In particular, for collections containing genomes of multiple strains of the same species, it is possible that certain groups of strains are more similar to each other than to the remaining strains, since some strains may be evolutionarily closer than others. Therefore a collection may contain clusters of similar sequences, where the similarity between the sequences within a cluster is high, but the similarity of sequences across clusters is low. In such cases, conducting relative compression on a per-cluster basis may be more suitable.

In collections such as the Hemo and Infl collections introduced in Chapter 4, there may not be a sequence that can be used as a reference. Such collections contain highly similar sequences that are not necessarily homologous to most other sequence in the collection. Currently, relative compression cannot be used for compressing such general collections of DNA sequences. To better compress collections containing sequences from multiple strains of the same species, and to compress more general repetitive DNA sequence collections with a relative compressor, it may be beneficial to construct an artificial reference that captures the repeats in the collection.

In this chapter, we explore methods to construct an artificial reference that con-
CHAPTER 8. REFERENCE CONSTRUCTION

tains all or most of the repeats present in the collection. Such a reference will better represent the sequences in the collection. Since dictionary compressors implicitly or explicitly construct a dictionary of repeats, in Section 8.2, we construct references using dictionaries generated by the compressors we analysed in Chapter 4, namely, Re-pair, Sequitur and dna-x, and our COMRAD compressor from Chapter 5. Then, using RLZ, we experimentally evaluate the references constructed using the repeats detected by these compressors. As the use of two compression algorithms, one to detect the repeats to construct the reference and another to conduct the relative compression, is wasteful, we then discuss some heuristics to extract repeats more efficiently from a collection. We begin with an analysis of the effects on compression when various sequences from a collection are chosen as a reference.

8.1 Reference selection

Since the reference string is central for relative compression, we analyse the compression results when various sequences from a collection are chosen as the reference. For this experiment, we used the Ecol, Scere, and Spara collections from DASET-REP introduced in Chapter 4. Each collection was compressed multiple times, using optimised RLZ, each time using a different sequence from the collection as the reference. The variation in the compressed collection sizes when the reference is varied is displayed in Figure 8.1 for the three collections.

For the Ecol collection, using the genome for the Sakai strain as the reference produces slightly better compression results than using the K12 strain, which we used as a reference in the experimental results of earlier chapters. Most of the strains in the collection can be chosen as the reference without leading to adverse compression results. However, if either of the strains SMS.3.5 and 07_K1 are chosen as the reference, then the compression results are worse by a few megabytes. The strains ATCC.8739, ATCC.33849 and BL21.DE3.IG do not appear to be very similar to the remaining strains, since choosing one of these strains as a reference leads to almost a doubling of the compressed size. Similar observations can be made with the Spara collection, where only a certain few sequences lead to significantly worse compression results. On the other hand, for the Scere collection, the differences in compressed sizes are not great regardless of the sequence chosen as the reference.

The above experiment shows that choosing a random sequence from the collection to be the reference may result in unsatisfactory compression results. The results also show that using even the reference genome of the species may not lead to the best possible compression results. A simple method to choose the best performing reference is to follow the above procedure: compress a collection multiple times using each sequence in the collection as the reference, then choose the sequence that produced the best result as the reference. However, for collections containing many sequences, or even a few large genomes, this method is costly. It may be possible to
8.1. REFERENCE SELECTION

(a) Compressed size of \textit{Ecol} collection vs reference string used

(b) Compressed size of \textit{Spara} collection vs reference string used

(c) Compressed size of \textit{Scere} collection vs reference string used

Figure 8.1: The compressed sizes of the collections \textit{Ecol}, \textit{Spara} and \textit{Scere}, in Mbytes, when each collection is compressed using every sequence in the collection as the reference.
use heuristics that measure the similarity between the sequences to determine the sequence that best represents the collection. However, such an approach will most likely require pair-wise comparison between sequences, which will also be costly.

As discussed earlier, in certain collections of DNA sequences, a single sequence may not represent the repetitions in all the sequences. For example, for collections containing genomes from multiple strains of the same species, certain groups of strains may be more closely related evolutionarily than other groups of strains. Therefore, the genomes may form clusters of similar sequences. To test this hypothesis, we plotted the starting positions of factors generated by RLZ that form alignments to the reference genome for the Spara and Scere collections, which are presented in Figure 8.2. The most interesting factor alignments are in the Spara collection, where three clusters of strains are visible. The strains N_43, N_44, N_45, and IF01804 mostly share the same factor start positions, so probably belong to one cluster. The strains A4, A12, DBVPG6304, UFRJ50791, UFRJ50816, YPS183, and UWOPS91.917_1 seem to belong to the second cluster, while the remaining strains belong to the third cluster. Interestingly, the clusters could also have been predicted using Figure 8.1. For the Scere collection, it seems that the strains do not belong to specific clusters and most strains share the same factor start positions.

To better compress such collections, it may be possible to determine the sequence clusters, and each cluster can be compressed independently with a relative compressor. Even if the clustering information is readily available, there is still the need to store multiple references, which may be expensive, since single sequences tend not to be compressible. Also, the efficiency of substring or whole sequence retrieval from a collection depends on the manner in which the multiple references are indexed for fast access. Since exploration of clustering algorithms is not the topic of this thesis, we do not pursue in this direction.

Deorowicz and Grabowski [2011], who proposed various improvements to the RLZ algorithm, suggested three ways to improve the reference string for their GDC algorithm. The first was to adjust the composition of the reference during compression. When substrings of a certain minimum length not occurring in the reference are encountered, they are appended to the reference, so that later occurrences of those substrings can be encoded as factors. Their results show that such a mechanism can provide a slight improvement to compression with minimal effects on the compression or decompression speeds. This method ensures that any repeats that are not already present in the reference are added to the reference. The downside is that it may over-compensate and add more substrings to the reference than necessary. Their second improvement is to use a simple heuristic to choose the best reference, which is the sequence that contains the maximum number of distinct 13-mers not containing any n nucleotides. Their results do not show any improvements as a result of using this approach. The final improvement they made is to allow the use of multiple references. Substring matches can be found in multiple references, but in
8.1. REFERENCE SELECTION

(a) Spread of factors across *S. paradoxus* reference

(b) Spread of factors across *S. cerevisiae* reference

Figure 8.2: The position components of some RLZ factors that form alignments between each sequence in the Spara and Score collections against the respective reference genomes for the species. The x-axis contains the position on the reference and the y-axis contains the sequence names.

In general, the initial reference is favoured over the additional references. Each reference is also encoded with respect to the first reference. Their results show that this variant significantly reduces the compression speed compared to the other variants, but the compressed sizes are also much better. We experiment with the various options of the GDC algorithm in Section 8.2.1.

We propose an alternative method, which is to artificially construct a reference that captures the repetitions in the collection. In the next section, we examine the use of a dictionary compressor to construct a reference for relative compression.
8.2 Dictionary compressors for reference construction

Recall from Section 3.2.2 that dictionary compressors identify repetitions occurring in the input and compress them using pointers to earlier occurrences. Adaptive dictionary compressors encode repeats using position and length pairs to refer to an earlier occurrence, while semi-static compressors add repeats to a dictionary and replace all occurrences of a repeat with a pointer to the dictionary entry. Since dictionary compressors explicitly detect repetitions, they are ideal for identifying repeats in a collection, so the dictionary of repeats generated by these compressors can be used to construct reference strings.

The aim is to detect repetitions occurring across the entire collection of sequences. Hence we chose the three dictionary compressors from Chapter 4 that can detect repeats globally, namely Re-pair, Sequitur, and dna-x, and our own algorithm COMRAD from Chapter 5. Re-pair and COMRAD are semi-static compressors, while Sequitur and dna-x are adaptive compressors. We hypothesise that the semi-static compressors would generate a more compact reference containing the necessary repetitions, but will be slower to generate and will require more memory. On the other hand, an adaptive compressor would generate a reference fast, using less memory, but, since the substitution choices are made in a greedy manner, is likely to generate a large reference containing redundant repeats.

For the semi-static compressors, the dictionary needs to be processed to extract the repeats and to remove any redundancy. For the two adaptive compressors, the repeats detected by dna-x need to be extracted from the substitutions made in the sequences, and the repeats detected by Sequitur need to be extracted from the context-free grammar. Redundant repeats also need to be carefully removed from the extracted repeats. Below, we briefly discuss each of these compressors, and how the output of the compression algorithms are processed to produce a reference string.

Re-pair: The Re-pair algorithm [Larsson and Moffat, 1999], discussed in Section 3.2.2, compresses the input by iteratively replacing all occurrences of the most frequent symbol pair with a non-terminal, until the input contains no symbol pairs with a count of more than one. The algorithm outputs a sequence of non-terminal symbols, and a dictionary of rules that map the non-terminals to the symbol pairs that they replaced. The dictionary is hierarchical, where one or both of the symbols in a pair could be a non-terminal. This dictionary contains the repeated substrings in the input, and these can be retrieved by expanding the right-hand sides of the rules. The substrings can then be concatenated to construct a reference. It is not necessary to add all expanded rules to the reference, since rules lower in the hierarchy are incorporated into the repeated substrings of rules higher in the hierarchy that refer to these rules. For example, expanding rule Z in the set of rules Z ← PQ, P ← aA, Q ← CD, results in rules P, Q, A, C, and D being expanded. Once
8.2. DICTI ONARY COMPRESSORS FOR REFERENCE CONSTRUCTION

$Z$ is expanded, it is redundant to expand the other non-terminals. To reduce this redundancy, we ensure that each repeat added to the reference is distinct.

The non-terminals generated by Re-pair are identified using unique integers. The higher the non-terminal number, the later the rule was generated, and the higher up in the hierarchy the rule is likely to be. Thus, we process rules from highest to lowest number, and expand a rule $Z$ if and only if $Z$ has not been expanded by a previous rule. If rule $Z$ is expanded, then the resulting substring is appended to the reference. This continues until all of the rules have been considered. The concatenation of the expanded substrings forms the reference.

COMRAD: Similar to Re-pair, COMRAD is a dictionary compression algorithm that detects repeated substrings in the input, and encodes them efficiently to achieve compression. COMRAD also operates in multiple iterations, however, it is a DNA-specific disk-based algorithm designed to compress large DNA datasets, as described in Section 5.2. The dictionary of substitution rules output by COMRAD is very similar to the Re-pair dictionary, so we use the same approach as for Re-pair.

Sequitur: Unlike Re-pair and COMRAD, Sequitur [Nevill-Manning and Witten, 1997b,c] is an adaptive dictionary compression algorithm, that constructs a context-free grammar that describes the input. The special non-terminal $S$ represents the input. As each symbol is appended to $S$, if the algorithm detects a repeated symbol pair, it is replaced by a new non-terminal if it is a new symbol pair, or an existing non-terminal if the symbol pair had already been detected earlier. After all symbols are read, the input string is compactly represented by rule $S$ as a series of terminals and non-terminals. The remaining rules in the grammar represent the redundancy that was detected. Section 3.2.2 describes the Sequitur algorithm in detail. As for Re-pair and COMRAD, the substitution rules produced by Sequitur are hierarchical. Therefore, the same approach used for Re-pair is used to generate a reference using the context-free grammar generated by Sequitur.

dna-x: Like Sequitur, dna-x [Manzini and Rastero, 2004] is an adaptive dictionary compression algorithm, as discussed in Section 3.3.1. It follows an LZ77-type approach, where repeated substrings are substituted as references to earlier occurrences of the substring. The repeats detected have certain length restrictions based on a substring length specified as an input to the algorithm. We use the repeated substrings detected by dna-x to construct the reference. However, unlike the semi-static compressors, dna-x does not uniquely identify multiple occurrences of the same repeat. If we were to naïvely concatenate the substrings represented by the replacements dna-x made, it is likely that more than one occurrence of the same repeat is added to the reference, making it unnecessarily large.

We modified the dna-x implementation to output the start and end position of
the earlier occurrence of each repeat, which we identify as an interval, followed by
the repeated substring. We use the intervals to avoid adding a repeat to the refer-
ence more than once. First, the intervals are sorted based on their start positions.
Then an interval at a time is read, and it is compared to the previous interval. If
the current interval does not overlap with the previous interval, then the substring
represented by the current interval is added to the set of substrings to be included
in the reference. The previous interval is then set to the current interval. Otherwise,
if the current interval is contained completely within the previous interval, then the
substring represented by that interval is ignored, as it was already added to the
reference when the previous interval was discovered. If the current interval overlaps
with the previous interval, then the suffix not contained in the previous interval
is concatenated to the previous substring that was added to the set, and then the
previous interval is extended to cover the current interval entirely. In this manner,
we ensure that the same repeat is not added more than once to the reference.

This method does not necessarily remove all redundant repeats, since the same
repeat may be referenced from different locations. Removing the redundancy using
the intervals will not capture such occurrences, since the start and end positions
of the intervals will be different. Therefore we take a further step. The substrings
output in the previous step are sorted alphabetically. Then we remove any substrings
that occur as a prefix of any other substrings. Even after this step, it is possible that
some redundant repeats still exist in the set, but we do not attempt to remove any
further redundancy so as not to increase the complexity of the reference construction.
The set of substrings remaining after removing the redundancy using the above two
steps are concatenated to construct the reference for relative compression.

In this manner, we construct reference strings using the repeats in the dictio-
naries produced by the chosen dictionary compressors. We expect the artificially
constructed references to be much larger than the individual sequences in the collec-
tion, since it contains repeats occurring across the entire collection. The reference
may also include some redundant repeats. Note that using a constructed reference
does not alter the operation of a relative compressor in any way. The only difference
is that an additional sequence is included as part of the collection. Next we use the
constructed references to test the compression performance of a relative compressor
for several collections of DNA sequences.

8.2.1 Experimental evaluation

We now test the compression performance when an artificial reference is used for rel-
ative compression. First we describe the experimental setup, then discuss the results
for compressing collections that do and do not contain an appropriate reference.
Test data and environment

To test the effectiveness of the reference strings constructed using the Re-pair, COMRAD, Sequitur, and dna-x dictionaries, we used the three collections Ecol, Spara, and Scere from dataset-rep of Chapter 4. The collections contain 33, 36, and 39 genomes of various strains from the species E. coli, S. paradoxus, and S. cerevisiae, respectively. If using a constructed reference leads to better compression than using the best available sequence in the collection as the reference, then some sequences in the collection were not well represented by the original reference.

Another aim of constructing an artificial reference is to compress collections for which choosing any of the sequences from the collection as a reference will lead to poor compression. Three such collections can be found in dataset-rep, namely Hemo, Infl, and Mito. We first show that relative compression using a single sequence as the reference produces poor compression results for these three collections. Then we use the constructed references to show the improvements in compression.

Re-pair, COMRAD, Sequitur, and dna-x were run on each of the six collections. Re-pair was run with the default parameters, without limiting the number or length of repeats that can be discovered. The implementation was modified to output the substitution rules in plain text so that they can be used to construct the reference. COMRAD was run with the starting substring length $L = 16$, and a minimum threshold frequency $F = 2$. COMRAD outputs the substitution rules in plain text before the rules are encoded for storage, so we used the plain-text dictionary output to construct the reference. For Sequitur, the default parameters were used without placing any restrictions on the number or length of repeats that can be discovered. The hash table memory limit was set to 2 Gbyte. The implementation has an option to output the substitution rules instead of the compressed output, and we used this output to construct the reference. For dna-x, the source code was modified to output just the intervals and the repeats, which was processed to produce the reference. For the algorithm parameters, we used a substring length $B = 16$ to be consistent with COMRAD. For $B < 16$, the better compression results tend to be offset by the compressed size of the larger reference. For $B > 16$, the compressed size of the smaller reference tends to be offset by the poorer compression results. Therefore, $B = 16$ is a reasonable value to capture the significant repeats.

We chose RLZ as the relative compressor for the experiment, due to its fast compression and decompression speed, as well as its low memory use. With the artificially constructed reference strings, we used the non-greedy factorisation option that factorises a sequence by detecting the longest factors in a region (Section 6.2.1), and the short factor encoding option (Section 6.2.2). The LISS factor encoding option (Section 6.2.3) was not used, since the reference is no longer a homologous sequence to the remaining sequences of the collection. The libdivsufsort library was used to generate the suffix array, and the the lcpdc algorithm [Puglisi and Turpin, 2008]
was used to generate the LCP array. The reference was compressed using 7-Zip with the options `-t7z -m0=lzma -mx=9 -mfb=64 -md=30 -ms=on`. Note that any relative compressor can be used for this experiment. Although we would have also liked to experiment with the XMCompress relative compressor, it was too slow to run multiple times as required to test each of the constructed reference strings.

The compression performance is measured in terms of the compressed size, compression and decompression time, and maximum memory usage during compression, as described in Section 4.2.1. The compressed size is measured as the total megabytes required to store the compressed collection and the compressed reference. The compression speed is measured as the total number of seconds required to generate the reference and to compress the collection using RLZ. The decompression speed is also measured in seconds, and is the time taken to decompress the reference and the compressed collection. The approximate maximum memory use is measured using the valgrind massif tool and the results reported are for the RLZ compression step. We also discuss the memory use during construction of the reference.

All experiments were conducted on a machine with four 2.6GHz Dual-Core AMD Opteron processors and 32 Gbytes of RAM running Ubuntu 8.04, using a single CPU.

**Results for compressing collections with a reference**

Table 8.1 presents the RLZ compression results for the Ecol, Spara, and Scere collections, using artificial references constructed using the Re-pair, COMRAD, Sequitur, and dna-x repeat dictionaries. For all three collections, using a constructed reference led to better compression results than using a single genome as the reference. The compressed size of the Ecol collection is more than halved. On the other hand, strains in the Scere collection, must have a much higher level of similarity, as the improvements from using a constructed reference is not so significant.

The compression speed, which includes the time taken to generate the reference, is a few times slower than using a sequence from the collection as a reference. For collections containing strains that are quite different from each other, such as the Ecol collection, constructing a reference seems to be worthwhile, but for a collection such as Scere, and to a lesser extent, Spara, the compression gains may not outweigh the slow compression speed. Decompression speeds were not affected by using a constructed reference. Therefore, if decompression is more frequent than compression, then the extra time required to compress may be worthwhile.

When comparing the construction techniques, using a Re-pair constructed reference produced the best compression results. Using a COMRAD constructed reference produced slightly worse results than Re-pair. This is expected, since Re-pair can detect repeats of any length, while COMRAD only detects repeats of length at least $L$. Overall, the two semi-static compressors produced better compressed sizes by around 1–2 Mbytes compared to the two adaptive compressors. This is also expected, since
the semi-static compressors carefully select the substitutions to be made, favoring more frequent repeats over less frequent ones. The adaptive compressors use a greedy approach to selecting repeats to substitute, hence are likely to make less optimal substitution decisions, resulting in a larger dictionary.

Out of the adaptive compressors, using a dna-x constructed reference produced slightly better compressed sizes than using a Sequitur constructed reference. Since Sequitur makes substitutions incrementally for a given repeat, it is possible that sub-standard substitution decisions are made that prevent long repeats from being extended. On the other hand, each substitution made by dna-x is for a complete repeat so the dna-x reference is likely to contain longer repeats than the Sequitur reference. This hypothesis is confirmed in the results. Sequitur parsed the Scere collection into nearly twice the number of factors as did dna-x, with similar results for the Spara collections. In fact, using a dna-x reference tends to produce similar number of factors to those produced using the Re-pair or COMRAD reference, and for the Spara collection, dna-x produced less factors than when a Re-pair or COMRAD reference was used. The main cause of the higher compressed sizes from using a dna-x reference is due to the higher factor position encoding cost associated with the longer reference. Therefore, further attempts to reduce the redundancy in the dna-x generated reference is required. Overall, constructing a reference using dna-x is faster than using the other compressors, although extracting repeats is complex.

When a constructed reference is used, the memory use of RLZ increased significantly. During compression and decompression, and for random access, RLZ stores the reference string in memory using $n_1 \log |\sigma'|$ bits, where $n_1$ is the length of the reference, and $\sigma'$ is the DNA alphabet, including the nucleotide $n$. Therefore, when using a constructed reference, the memory use of RLZ is significantly larger due to the larger reference size, and the larger suffix array and other associated data structures, which are stored in memory. Memory use during reference construction also needs to be considered. In this phase, dna-x used less memory than during the RLZ compression step. Sequitur used approximately 2 Gbytes, which is for the hash table. Re-pair used the most significant amount of memory, requiring approximately 12 Gbytes for the two yeast collections. COMRAD on the other hand only used approximately 1.5 Gbytes during reference construction. Therefore, Re-pair is the least memory-efficient method for constructing a reference, despite the best results.

Table 8.2 shows a break-down of the compressed sizes. A constructed reference is significantly larger than an individual sequence in a collection. However, when the references are compressed using 7-Zip, the compressed sizes are only a few megabytes larger than a compressed reference from the collection. In comparison, when using a constructed reference, the compressed collection sizes are significantly smaller. Therefore, the increased compressed reference size tends to be offset by the reduction in the compressed sequence sizes.

Table 8.2 also shows a break-down of the compression times. When using the
<table>
<thead>
<tr>
<th>Collection</th>
<th>Algorithm</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decompress. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>RLZ-opt</td>
<td>17.82</td>
<td>0.91</td>
<td>200</td>
<td>71.84</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>7.36</td>
<td>0.37</td>
<td>622</td>
<td>428.02</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>7.67</td>
<td>0.39</td>
<td>600</td>
<td>456.00</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>8.96</td>
<td>0.46</td>
<td>599</td>
<td>518.99</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>8.71</td>
<td>0.44</td>
<td>566</td>
<td>756.25</td>
<td>2</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ-opt</td>
<td>12.62</td>
<td>0.25</td>
<td>276</td>
<td>138.70</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>7.35</td>
<td>0.14</td>
<td>1,554</td>
<td>1,043.95</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>7.73</td>
<td>0.15</td>
<td>1,779</td>
<td>1,094.49</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>9.15</td>
<td>0.18</td>
<td>1,640</td>
<td>1,277.72</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>8.39</td>
<td>0.16</td>
<td>1,499</td>
<td>1,822.52</td>
<td>4</td>
</tr>
<tr>
<td>Scere</td>
<td>RLZ-opt</td>
<td>8.90</td>
<td>0.15</td>
<td>229</td>
<td>131.06</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>6.18</td>
<td>0.11</td>
<td>1,672</td>
<td>1,264.32</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>6.46</td>
<td>0.11</td>
<td>1,775</td>
<td>1,299.18</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>9.31</td>
<td>0.16</td>
<td>1,882</td>
<td>1,463.25</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>7.16</td>
<td>0.12</td>
<td>1,529</td>
<td>2,155.40</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 8.1: RLZ compression results for three collections from DATASET-REP. The columns are: collection name, algorithm used to generate the results, compressed size in Mbytes, average number of bits per base used, compression time in seconds including the time to construct the reference, approximate maximum memory used during RLZ compression in Mbytes, and decompression time in seconds.

Comparing the compression results to the best results from Chapter 4 shows that using a constructed reference, combined with RLZ, produced better results than two of the best compressors, Sequitur and Re-pair, as shown in Figure 8.3. In other words, using a Re-pair or Sequitur constructed reference, then compressing the collections with RLZ, produced better results than simply compressing with Re-pair or Sequitur by around 2–4 Mbytes for Re-pair and 1–2 Mbytes for Sequitur, but at the cost of slower compression. Also, using a COMRAD constructed reference with RLZ the compressed sizes were almost halved compared to compressing the same collection with COMRAD. However, using a constructed reference still did not produce better results than using 7-Zip, or the DNA compressors, dna-x and XMCompress.

The results show that using an artificial reference constructed from repeats detected by various dictionary compressors produces excellent results. So far, we have only experimented with collections which already contained an appropriate reference. Next we evaluate the compression performance when collections that do not contain an appropriate reference are compressed with an artificial reference.
8.2. DICTIONARY COMPRESSORS FOR REFERENCE CONSTRUCTION

<table>
<thead>
<tr>
<th>Collection</th>
<th>Algorithm</th>
<th>Orig. ref. size (Mbases)</th>
<th>Comp. ref. size (Mbytes)</th>
<th>Comp. seq. size (Mbytes)</th>
<th>Ref. constr. time (secs)</th>
<th>RLZ comp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>RLZ-opt</td>
<td>5.50</td>
<td>1.30</td>
<td>16.51</td>
<td>—</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>42.53</td>
<td>3.89</td>
<td>3.47</td>
<td>392</td>
<td>230</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>43.43</td>
<td>3.94</td>
<td>3.75</td>
<td>352</td>
<td>248</td>
</tr>
<tr>
<td></td>
<td>RLZ-Seqitur</td>
<td>51.95</td>
<td>4.61</td>
<td>4.34</td>
<td>335</td>
<td>454</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>73.79</td>
<td>4.53</td>
<td>4.18</td>
<td>196</td>
<td>370</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ-opt</td>
<td>12.00</td>
<td>2.82</td>
<td>9.80</td>
<td>—</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>102.47</td>
<td>4.20</td>
<td>3.15</td>
<td>901</td>
<td>775</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>107.52</td>
<td>4.35</td>
<td>3.37</td>
<td>1,154</td>
<td>641</td>
</tr>
<tr>
<td></td>
<td>RLZ-Seqitur</td>
<td>125.82</td>
<td>5.28</td>
<td>3.86</td>
<td>860</td>
<td>940</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>174.01</td>
<td>5.28</td>
<td>3.11</td>
<td>496</td>
<td>1,035</td>
</tr>
<tr>
<td>Scere</td>
<td>RLZ-opt</td>
<td>12.16</td>
<td>2.86</td>
<td>6.04</td>
<td>—</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>124.46</td>
<td>3.92</td>
<td>2.26</td>
<td>994</td>
<td>867</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>127.95</td>
<td>4.05</td>
<td>2.41</td>
<td>1,074</td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>RLZ-Seqitur</td>
<td>139.35</td>
<td>5.64</td>
<td>3.67</td>
<td>998</td>
<td>1,081</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>206.09</td>
<td>4.83</td>
<td>2.33</td>
<td>433</td>
<td>1,122</td>
</tr>
</tbody>
</table>

Table 8.2: The break-down of the compressed sizes and the compression time. The columns are: collection name, algorithm used to compress, reference size in Mbases, compressed reference size in Mbytes, compressed size of sequences in the collection in Mbytes, time taken to construct reference, and time taken for RLZ compression.

Figure 8.3: Comparison of the compressed sizes (in bpb) for the test collections.

Results for compressing collections without a reference

Reference sequence construction enables relative compressors to compress collections that do not contain a suitable reference. Examples of such collections are the
Figure 8.4: Comparison of the compressed sizes (in bpb) for the test collections.

**Hemo**, **Infl**, and **Mito** collections from **DATASET-REP**. The **Hemo** collection contains sequences that are associated with creating the **Hemo** protein. Multiple genes are associated with this protein and the collection contains sequences for the various genes. There is no single sequence that is homologous to all of the distinct genes. The same is true for the **Infl** collection, which contains sequences from many strains of the influenza virus and other DNA sequences that are associated with the virus. The **Mito** collection contains mitochondrial DNA from many different species and there is no single sequence that represents mitochondrial DNA. We now use **Re-pair**, **COMRAD**, **Sequitur**, and **dna-x** to construct a reference for each of these collections, so that a relative compressor can be used to compress them.

Results are presented in Table 8.3. The first row of results for each collection contains the compression results when a sequence from the collection is used as a reference string. Only the non-greedy factorisation option detecting local maximum factors, and short factor encoding option were used for **RLZ-opt**. Since no single sequence from each of these collections is a good reference string, we used the first sequence in the collection as the reference as a baseline. This led to worse results than using a general-purpose compressor. Using an artificial reference, the compression results can be improved significantly. However, the collections were not compressed as well as using **7-Zip** on the three collections, or as well as **Sequitur**, or **COMRAD**, for the **Hemo** and **Mito** collections, as shown in Figure 8.4. For the **Infl** collection, the compression results tend to be better with relative compression. Judging by the results from the previous section, it seems that using relative compression with an artificial reference produces better compression results for larger collections.
8.2. DICTIONARY COMPRESSORS FOR REFERENCE CONSTRUCTION

<table>
<thead>
<tr>
<th>Collection</th>
<th>Algorithm</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decomp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hemo</td>
<td>RLZ-opt</td>
<td>2.83</td>
<td>3.22</td>
<td>3</td>
<td>16.14</td>
<td>≪1</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>1.28</td>
<td>1.07</td>
<td>17</td>
<td>27.90</td>
<td>≪1</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>1.16</td>
<td>1.27</td>
<td>20</td>
<td>27.04</td>
<td>≪1</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>1.18</td>
<td>1.35</td>
<td>20</td>
<td>30.64</td>
<td>≪1</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>1.15</td>
<td>1.31</td>
<td>17</td>
<td>35.24</td>
<td>≪1</td>
</tr>
<tr>
<td>Mito</td>
<td>RLZ-opt</td>
<td>7.97</td>
<td>2.65</td>
<td>11</td>
<td>22.53</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>5.91</td>
<td>1.96</td>
<td>90</td>
<td>63.31</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>5.77</td>
<td>1.92</td>
<td>110</td>
<td>56.68</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>6.36</td>
<td>2.11</td>
<td>77</td>
<td>63.38</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>5.77</td>
<td>1.92</td>
<td>77</td>
<td>84.55</td>
<td>1</td>
</tr>
<tr>
<td>Infl</td>
<td>RLZ-opt</td>
<td>37.13</td>
<td>2.77</td>
<td>45</td>
<td>53.78</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>2.69</td>
<td>0.20</td>
<td>296</td>
<td>318.80</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-COMRAD</td>
<td>2.86</td>
<td>0.21</td>
<td>242</td>
<td>323.69</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-Sequitur</td>
<td>3.43</td>
<td>0.26</td>
<td>312</td>
<td>318.80</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RLZ-dna-x</td>
<td>3.46</td>
<td>0.26</td>
<td>336</td>
<td>485.68</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.3: RLZ compression results for collections typically unsuitable for relative compression. The columns are: collection name, algorithm used to compress, compressed size in Mbytes, average number of bits per base used, compression time in seconds including the time to construct the reference, approximate maximum memory used during RLZ compression in Mbytes, and decompression time in seconds.

We have shown that using an artificial reference for relative compression can produce better results for collections that do not contain a single sequence that can be used as a reference. However, better results can be produced by using a compressor such as 7-Zip or Sequitur. COMRAD also produced better results than this method, and, like RLZ, COMRAD also preserves sequence boundaries and permits random access queries on the compressed set. Although the results may be better for larger collections, constructing a reference for such collections with current methods will not be practical due to slow compression speed and high memory use.

Comparison of artificial reference construction to GDC

We now compare the best results from our reference construction experiment with the GDC implementation with improved reference string options. There are five alternatives: GDC-normal, which is the standard implementation that improves upon RLZ; GDC-R, which uses a heuristic to choose the best reference available in the collection; GDC-advanced, which appends additional phrases to the reference, GDC-advanced-R; which is the combination of the two previous improvements; and GDC-ultra, which uses multiple references. For GDC-advanced, we set the option -ma=32 so that substrings of length 32 not occurring in the reference are appended to the reference. For GDC-ultra, the advanced and R options were turned off, and the number of references to use was specified as 40. These are the parameter values used by the authors
in their experiments. The remaining parameters are left as the default values.

Table 8.4 shows the comparison between the best-performing RLZ-Re-pair variant, and the GDC variants that improve the reference string. The standard RLZ and GDC results are also included as RLZ-opt and GDC-normal. The GDC-R variant chose the O26_H11 strain as the best reference for the Ecol collection, which led to worse results than using the K12 strain. For the Spara and Scere collections, the reference selected is the reference genomes, so there is no improvement in the results. A few extra seconds are required to select the best reference. Overall, if the existence of a better reference can be checked using a few extra seconds, then it is a worthwhile option to use. The variant GDC-advanced, which appends extra substrings to the reference, produced significantly better results for Ecol, even better than with RLZ-Re-pair. The appended symbols amount to around 3 Mbytes compressed, but the compressed collection reduced in size by around 10 Mbytes compared to GDC-normal. However, for Spara and Scere, the improvements of GDC-advanced are modest, and worse than RLZ-Re-pair. The compressed additional substrings amount to less than 1 Mbyte, which disproves our initial hypothesis that such a technique would lead to the reference unnecessarily increasing in size.

Combining GDC-R and GDC-advanced led to worse compression results for Ecol. It seems that the O26_H11 strain is not appropriate when further substrings are to be appended to the reference. The compression results are unchanged for Spara and Scere, since the heuristically chosen reference is the same as the reference we specified for GDC-advanced. Overall, none of the GDC variants discussed so far achieved better compression results than RLZ-Re-pair. The most significant improvement in compression is observed when multiple references are used. The compression speed is significantly slower compared to the other GDC variants, but it is significantly faster than RLZ-Re-pair and leads to much better compression results. If multiple references can be managed carefully, as is done by GDC, then significantly better compression results can be achieved.

In terms of the decompression speed, all GDC variants are competitive with RLZ. In terms of memory use, most GDC variants have memory usage similar to RLZ-opt. GDC-advanced tends to use more memory than GDC-normal, since the reference with appended substrings is larger. GDC-ultra uses approximately double the memory used by RLZ-Re-pair. However, this memory use is much less than the memory used by Re-pair during reference construction, therefore, even with its high memory use, GDC-ultra still scales for compressing large collections.

The approaches taken by Deorowicz and Grabowski [2011] for improving the quality of the reference string are efficient and produce excellent compression results. However, we aim to explore ways to efficiently construct an appropriate reference. Next we analyse the practical implications of constructing reference strings, and also discuss some heuristics that could be used to detect repeats in a collection without having to compress the entire collection with a dictionary compressor.
8.2. DICTIONARY COMPRESSORS FOR REFERENCE CONSTRUCTION

<table>
<thead>
<tr>
<th>Collection</th>
<th>Algorithm</th>
<th>Comp. size (Mbytes)</th>
<th>Bits per base</th>
<th>Comp. time (secs)</th>
<th>Comp. mem. (Mbytes)</th>
<th>Decomp. time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecol</td>
<td>RLZ-opt</td>
<td>17.82</td>
<td>0.91</td>
<td>200</td>
<td>71.84</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>7.36</td>
<td>0.37</td>
<td>622</td>
<td>428.02</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>13.72</td>
<td>0.70</td>
<td>64</td>
<td>80.45</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-R</td>
<td>14.50</td>
<td>0.74</td>
<td>80</td>
<td>85.93</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced</td>
<td>7.34</td>
<td>0.37</td>
<td>37</td>
<td>172.80</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced-R</td>
<td>7.51</td>
<td>0.38</td>
<td>40</td>
<td>186.65</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>GDC-ultra</td>
<td>5.54</td>
<td>0.28</td>
<td>234</td>
<td>1,149.83</td>
<td>2</td>
</tr>
<tr>
<td>Spara</td>
<td>RLZ-opt</td>
<td>12.62</td>
<td>0.25</td>
<td>276</td>
<td>138.70</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>7.35</td>
<td>0.14</td>
<td>1,554</td>
<td>1,043.95</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>8.77</td>
<td>0.17</td>
<td>24</td>
<td>136.28</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-R</td>
<td>8.77</td>
<td>0.17</td>
<td>26</td>
<td>136.73</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced</td>
<td>8.35</td>
<td>0.16</td>
<td>19</td>
<td>157.74</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced-R</td>
<td>8.35</td>
<td>0.16</td>
<td>23</td>
<td>135.31</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-ultra</td>
<td>4.77</td>
<td>0.09</td>
<td>134</td>
<td>3,005.48</td>
<td>3</td>
</tr>
<tr>
<td>Scere</td>
<td>RLZ-opt</td>
<td>8.90</td>
<td>0.15</td>
<td>229</td>
<td>131.06</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>RLZ-Re-pair</td>
<td>6.18</td>
<td>0.11</td>
<td>1,672</td>
<td>1,264.32</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-normal</td>
<td>6.55</td>
<td>0.11</td>
<td>15</td>
<td>132.25</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-R</td>
<td>6.55</td>
<td>0.11</td>
<td>19</td>
<td>141.40</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced</td>
<td>6.34</td>
<td>0.11</td>
<td>14</td>
<td>143.02</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>GDC-advanced-R</td>
<td>6.34</td>
<td>0.11</td>
<td>17</td>
<td>141.40</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>GDC-ultra</td>
<td>4.34</td>
<td>0.07</td>
<td>192</td>
<td>3,361.91</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 8.4: RLZ and GDC compression results for three collections from DATASET-REP. The columns are: collection name, algorithms used to compress, compressed size in Mbytes, average number of bits per base used, compression time in seconds including the time to construct the reference, approximate maximum memory used during compression in Mbytes, and decompression time in seconds.

8.2.2 Discussion

Earlier we showed that a single reference may not represent all the sequences in a collection. We proposed the construction of an artificial reference using repeats detected by dictionary compressors. Using these artificial references, the compression results improved for three collections for which an adequate reference was available, and three other collections which did not contain a sequence that could be used as a reference. This improvement was expected, since a reference constructed in such a manner is likely to contain the repeats present in the entire collection, although the overhead of the larger reference could potentially offset any compression gains.

A main disadvantage of constructing a reference in this manner is higher memory use. When RLZ uses a reference from the collection, compression is fast and memory use is low. Using an artificial reference, which is usually a few times larger than a reference chosen from the collection, memory use is around ten times larger. The memory overhead can be reduced by storing the reference, suffix array and other
CHAPTER 8. REFERENCE CONSTRUCTION

associated data structures in a compressed form at the cost of compression speed.

Another disadvantage of constructing a reference in this manner is slower compression. Using semi-static compressors, constructing the reference is slow, but RLZ compression is faster since the reference is compact. Using adaptive compressors, it is faster to construct the reference, but due to the larger reference size, RLZ compression is slower. Therefore, if the reference needs to be constructed rarely, then using a semi-static dictionary compressor may be feasible. Once the reference is constructed, other sequences that were not in the collection during reference construction can also be compressed with the same reference, provided the new sequences are highly similar to at least two sequences already in the collection. If the reference needs to be reconstructed regularly, then using dna-x is better due to the low resource use.

Overall, using two compressors, one to construct a reference and the other to perform relative compression will not scale for large collections. Restrictions can be placed on the dictionary compressors we chose to improve memory use and compression speed, but it is unclear what impact this will have on compression. We leave the exploration of these parameters as a future exercise.

As shown in Section 8.2.1, the approaches adopted by Deorowicz and Grabowski [2011] are faster than ours. Their simplest approach does not require an extra pass through the collection to append potential repeats to the reference, and our experiments show that this method tends not to add more substrings to the reference than necessary. However, our approach ensures that the reference only consists of repeats, and the results of using RLZ-Re-pair tends to be better than the GDC variants that modify the reference. The most significant contribution made by Deorowicz and Grabowski [2011] is the algorithm that allows multiple references to be used. This variant was implemented in a manner which is fast and memory-efficient, while also producing significantly better compression results. It is promising to see that a practical method exists for improving the compression performance with low overhead.

There are two main drawbacks. The first is that an artificially constructed reference can be used by any relative compressor, while the approach used by GDC requires that each relative compressor be modified to manage multiple references. The second drawback is in adding the display() functionality, as it is not trivial to manage multiple references to provide access to substrings in the compressed collection.

Several techniques can be used to reduce the time and memory requirements for constructing a reference. One is to construct a reference using a sample of sequences from the collection, rather than using the entire collection. Sequences could be randomly sampled, or, if information is available regarding the similarity of the sequences in a collection, more informed judgements can be made on choosing sequences to cover as many of the repeats as possible.

Another option is to use an existing repeat detection tool that does not have the added complexity of being a compression algorithm. Several mature tools for repeat detection in DNA sequences exist. The RepeatMasker tool masks any interspersed
and low complexity repeats in DNA sequences using a database of known repeats. Its purpose is to mask repeats, particularly in non-coding regions, so that detecting genes and other coding regions would return more accurate results. Another tool, RepeatScout [Price et al., 2005], operates in a similar manner to COMRAD, where initially, the counts of all L-mers are recorded. The L-mers and their counts are used to determine the repeats that occur in the sequence. We attempted to use this tool to construct a reference for the Ecol collection, but, given that the program did not end after an hour of execution, we deemed it unsuitable for reference construction. Another tool, RECON [Bao and Eddy, 2002], uses pair-wise alignments between sequences in a collection to identify repeat families. However, obtaining pair-wise alignments between large numbers of sequences is infeasible. Another option is to align the concatenated sequences in a collection against itself using blastn [Zhang et al., 2000], then extract the substrings in high scoring alignment regions to construct the reference. Once again, this is time consuming and infeasible for large collections. Overall, these repeat detection tools do not scale for large collections.

Next we briefly explore some of our own heuristics for fast repeat detection.

8.3 Heuristics for repeat detection

As seen in our experiments, dictionary compressors, especially semi-static ones, are capable of producing a compact reference containing the necessary repetitions. However, the time taken to construct such a reference is excessive. Therefore, we explore two heuristics to determine the repeats in a collection. The first heuristic is based on COMRAD, and the second is based on repeat detections using a suffix array.

COMRAD heuristic

Since COMRAD in its first iteration substitutes L-mers with a frequency of at least $F$ that do not overlap with other more frequent L-mers, COMRAD effectively detects all significant repeats during this iteration. The remaining iterations of COMRAD simply substitute repeated sets of non-terminals to reduce the encoding cost. Instead of using multiple iterations, we used the first iteration to estimate repeated substrings that may be detected by COMRAD as repeats. First, we calculated the frequencies of all 16-mers. Then with a single pass through the sequences we output substrings which contained overlapping L-mers with a frequency of at least two. We were careful to check for reverse complements and to ensure that runs of reverse complements are detected correctly. The substrings were then sorted in alphabetical order and the frequencies of distinct substrings were calculated. Then distinct substrings with frequencies of at least two were output to produce a reference.

We used this reference to compress the Ecol, Spara, and Scere collections. The results were mixed. For the Ecol collection, the time taken to construct the
reference was 234 seconds compared to the 352 seconds required when using COMRAD to construct the reference. The compressed size was 16.55 Mbytes, with 4.92 for the reference and 11.63 for the collection. This is significantly higher than the 7.67 Mbytes that can be achieved by using a reference constructed using COMRAD.

For the Spara collection, the compressed size was 15.47 Mbytes, with 5.32 for the reference and 10.15 for the collection. For the Scere collection, the compressed size was 15.23 Mbytes, with 4.61 for the reference and 10.62 for the collection. For both these collections, the compressed size using the reference constructed by the heuristic is worse than the result that can be achieved by using the RLZ-opt algorithm. This experiment shows that the multiple passes made by semi-static dictionary compressors are important and attempting to approximate the repeat selection process leads to poor repeat detection.

**Suffix array heuristic**

The second heuristic uses the suffix array of the collection and its LCP array. Recall that the suffix array contains the indexes of the suffixes of a string sorted in lexicographic order, while the LCP array contains the length of the prefixes shared between adjacent suffixes in the suffix array. Adjacent elements in a suffix array tend to share a common prefix and these shared prefixes represent the repeats in a collection. However, certain repeated prefixes are suffixes of other repeated prefixes, and so care is required not to include redundant repeats. Therefore, we detect supermaximal repeats, which we define in terms of local maximum intervals of the LCP array. The following definitions are extracted from Abouelhoda et al. [2004].

**Definition 7.** Let $SA$ and $SA_{LCP}$ be the suffix and LCP arrays of a string. An $\ell$-interval of the LCP array is an interval $[i \ldots j]$, where $0 \leq i < j \leq n$, if

1. $SA_{LCP}[i] < \ell$,
2. $SA_{LCP}[k] \geq \ell$, for all $k$ in the range $i + 1 \leq k \leq j$,
3. $SA_{LCP}[k] = \ell$ for at least one $k$ in the range $i + 1 \leq k \leq j$, and
4. $SA_{LCP}[j + 1] < \ell$.

In other words, an $\ell$-interval of the LCP array is an interval with LCP values of at least $\ell$, and the LCP values on either side of the interval are less than $\ell$.

**Definition 8.** An $\ell$-interval is a local maximum interval in the LCP array if $SA_{LCP}[k] = \ell$ for all $k$ in the range $i + 1 \leq k \leq j$.

The prefixes of the suffixes represented by the local maximum interval of the LCP array correspond to substrings in a string that are repeated at least twice. However, these repeats could be contained within other repeats. Therefore, we
need to detect supermaximal repeats in a string, which are repeats that do not occur within other repeats in the same string. Below we define the properties of \( \ell \)-intervals that correspond to supermaximal repeats.

**Definition 9.** A string \( s \) is a supermaximal repeat if and only if there is an \( \ell \)-interval \([i \ldots j]\) such that

1. \([i \ldots j]\) is a local maximum in \( SA_{LCP} \) with \( \ell = |s| \), and
2. the symbols at \( S[SA[i] - 1], S[SA[i + 1] - 1], \ldots, S[SA[j] - 1] \) are pairwise distinct.

The two conditions ensure that the repeat \( s \) is not contained within any other repeats. Abouellhoda et al. [2004] proposed a linear time algorithm to find all supermaximal repeats in a string using the suffix and LCP arrays for the string. The algorithm passes through the LCP array, and when a local maximum interval is detected, the second condition in Definition 9 is checked to ensure that the repeat is not a suffix of a longer repeat. If the current local maximum interval does in fact correspond to a supermaximal repeat, then that repeat is output. The algorithm continues until the entire LCP array is processed.

We use this algorithm to determine the repeats in a collection. We used the Ecol, Spara, and Scere collections to evaluate the compression performance using a reference constructed with this suffix array approach. First we constructed the suffix array of an entire collection using the libdivsufsort implementation, and the LCP array of an entire collection using the lcpdc implementation. Then we used the above algorithm to output supermaximal repeats for each collection. For the Ecol collection, 192 seconds were required to construct the reference, and 306 seconds to compress with RLZ. The total compressed size was 10.08 Mbytes; 4.76 for the reference and 5.32 for the collection. For the Spara collection, 548 seconds were required to construct the reference, and 1,050 seconds to compress with RLZ. The total compressed size was 8.69 Mbytes; 4.67 for the reference and 4.02 for the collection. For the Scere collection, 671 seconds were required to construct the reference, and 976 seconds to compress the collection with RLZ. The total compressed size was 6.90 Mbytes; 4.20 for the reference and 2.70 for the collection.

Overall, the results were better than that could be achieved with the COMRAD heuristic and RLZ-opt by a few megabytes, but slightly worse than the results of RLZ-Re-pair due to the greater number of factors in the parsing. This was an unexpected result given that the set of supermaximal repeats should cover all repeats in the collection, hence the number of factors in the RLZ parsing should be lower. The repeats detected by these algorithms need to be further analysed to determine the cause of this unexpected result. The results are also comparable with that of using a dna-x generated reference, in terms of the compression speed and the compression results achieved. When comparing the results to those of GDC from Table 8.4, the
results are either comparable or better than GDC-normal, but tends to be slightly worse than GDC-advanced. Also the compression speed of GDC, including GDC-ultra, is much faster than using the suffix array heuristic.

The experiments show that using a suffix array to detect repeats and construct an artificial reference for a collection produces promising results. Construction of the suffix and LCP arrays is expensive and may not be feasible for larger collections, and is an active area of research. For very large collections, it may be possible to use disk-based methods for constructing these data structures.

8.4 Chapter summary

In this chapter, we showed that selecting an appropriate reference is an important decision when using a relative compressor. Choosing a reference that does not represent the sequences in the collection can lead to poor compression results. Also a single reference may not capture all the repeats present in the entire collection. Therefore, we proposed to construct an artificial reference that captures all or a majority of the repeats present in a collection. First we explored the use of various dictionary compressors to construct a reference, since dictionary compressors explicitly detect repeats during compression. We showed that, this approach leads to better compression, where at times the compressed sizes were more than halved compared to using a single reference. The price is slower compression, and higher memory use compared to using a sequence from the collection as a reference. Although the compression results are excellent, using a dictionary compressor to construct a reference is infeasible, especially for very large collections.

We therefore examined two heuristics for constructing references. One heuristic is based on the first iteration of the COMRAD algorithm, but the resulting reference did not produce promising compression results. The second heuristic used a suffix array of the collection and its LCP array to construct a reference, and achieved better compression. However, construction of suffix and LCP arrays for large collections will not scale, therefore more scalable alternatives should be explored.

We discussed the improvements made by Deorowicz and Grabowski [2011], who introduced several techniques for improving the quality of the reference. We found that while their compression results do not always meet the best results we achieved, their compression speed is significantly faster and the memory use is lower. Therefore, their techniques for improving the reference can scale better for large collections, as shown by their experiments with the collection of 70 human genomes.

Our results emphasise that better compression results can be achieved by using better reference strings in relative compression. The improvements are important for compactly storing large repetitive sequence collections, while preserving access to the compressed collections. The good results obtained by using a dictionary compressor to construct a reference were expected, given their repeat detecting
nature. The slow speed of running two compression algorithms to compress a single collection, and the high memory use due to the larger reference sizes were also expected outcomes. The experiments also provided a glimpse of the scalability issues of constructing references for large collections. Intuitively, constructing a reference using all sequences in a collection is unlikely to be feasible. It is most likely that some heuristic will be required to choose some of the sequences to base the reference on. Instead of constructing a reference, using multiple sequences in the collection as references is also an option, but once again, heuristics are required to select the references. Furthermore, constructing self-indexes based on multiple references will also be an interesting extension.

Improving the quality of the reference string for relative compression, and the related issue of constructing a suitable reference, remains open problems. Improvements to existing heuristics and new techniques need to be explored, especially in the compression speed and memory use fronts.
Chapter 9

Conclusion

The aim of our research project was to develop algorithms to assist researchers with managing the large quantities of data produced by DNA sequencing projects. In this thesis, we focused on algorithms for compressing large collections of assembled DNA sequences. We introduced two compression algorithms, COMRAD and RLZ, and also made modifications to enable substring extraction from compressed collections. RLZ was also modified to produce an index that allows for pattern searching in compressed collections, with some limitations. These algorithms can be used by researchers to compactly store large collections, while also allowing them to use the data while its compressed. For large sequence collections, the overhead of storing uncompressed data, or storing compressed data and then decompressing to access the data, is high. These issues can be overcome with the use of our algorithms. Below, we summarise the contributions made in this thesis and the future extensions for these algorithms. We also discuss further research topics in the field.

In Chapter 4 we conducted an experimental evaluation of publicly available general-purpose and DNA-specific compression tools for collection compression and single-sequence compression. The tools typically used by researchers to compress DNA sequences, namely gzip and bzip2, produced some of the worst compression results. However, compression tools such as 7-Zip and the implementations of Re-pair [Larsson and Moffat, 1999] and Sequitur [Nevill-Manning and Witten, 1997b,c], compressed collections very well, with 7-Zip being the best general-purpose compression tool for compressing collections and single sequences.

Out of the DNA compression algorithms, dna-x and XMCompress were the only implementations that scaled to compress most of the larger collections and sequences, and these produced the best compression results. The experimental evaluation emphasised the lack of scalability of some existing algorithms such as Re-pair, Sequitur, dna-x and XMCompress for large sequence compression, where certain collections had to be compressed in blocks. The main observation made was that most existing algorithms do not support collection compression. Sequences in a collection were concatenated into a single sequence before compression to enable global repeat
detection. This emphasised the need for collection compression algorithms that maintain sequence boundaries, while detecting repetitions across multiple sequences to achieve better compression, and also allow for individual sequence decompression.

In Chapter 5 we introduced our first collection compression algorithm COMRAD, a semi-static disk-based dictionary compression algorithm. At each iteration, COMRAD uses two passes over the input to ensure that all repetitions satisfying certain conditions are detected in the entire collection before substitution decisions are made. The algorithm also favours the substitution of highly frequent repeats over repeats occurring with a lower frequency. COMRAD is based on RAY [Cannane and Williams, 2001], therefore we first discussed the modifications made to RAY to make the algorithm suitable for compressing large collections of DNA sequences. Then the COMRAD compression and decompression algorithms were presented, followed by an experimental evaluation of our implementation. The results showed that COMRAD compressed our test collections well but was unable to produce better results than the best tools from Chapter 4, namely 7-Zip, dna-x, and XMCompress. The aim of detecting repetitions occurring across multiple sequences in a collection was achieved without pre-processing the input to make the repetitions visible to the algorithm, as we did for the existing compression tools. The sequence boundaries of the collection are also preserved and individual sequence extraction without decompressing the entire collection is possible. The COMRAD algorithm has two input parameters and the effects of changing these parameters were also discussed.

Next in Chapter 5, we discussed the modifications to the COMRAD-compressed output required to enable substring extraction from the compressed collection. We then discussed the display() query algorithm and compared the performance of our implementation of the query with that of RLCSA [Mäkinen et al., 2009; Mäkinen et al., 2010] and LZ-End [Kreft and Navarro, 2010, 2011]. The results showed that the COMRAD display query tends to be fast for extracting long substrings but slow for short substrings. Also, the COMRAD-index was larger than the RLCSA and LZ-End indexes. Overall, several improvements need to be made to the COMRAD compression algorithm before it can scale for compressing even larger collections, as will be discussed shortly.

We introduced our second compression algorithm in Chapter 6, which is suitable for compressing collections from resequencing projects containing individual genomes from the same species, where the genomes tend to be highly similar to a reference genome of the species. For such collections, it may be sufficient to compress each sequence with respect to a reference without the overhead of global repeat detection. This is the approach taken by RLZ. Unlike existing algorithms of this nature [Brandon et al., 2009; Christley et al., 2009], RLZ did not make any assumptions regarding the types of mutations present in each sequence with respect to the reference. We first discussed the basic RLZ algorithm, which used a Lempel-Ziv parsing for compression, then discussed three improvements that led to improved compression results. The first improvement was to use a non-greedy Lempel-Ziv
parsing instead of the greedy parsing used in the basic algorithm. The second and third improvements were two methods to improve the encoding techniques. The experimental evaluation of our implementation showed that standard RLZ produced good compression results, but with the optimised version, the compression results were almost halved for certain collections. However, the results of RLZ were not better than 7-Zip, dna-x, XMCompress or COMRAD, since the algorithm does not detect repeats globally. However, RLZ had the fastest compression speed, fast decompression speed, and very low memory use, compared to existing tools. Therefore, RLZ provides a good trade-off between compression and resource use.

In Chapter 7, we made some trivial modification to RLZ to support the display() query and compared the performance of our query implementation to those of COMRAD, RLCSA and LZ-End. The results showed that RLZ display() is the fastest, with the query speed improving for longer query lengths. The RLZ-index size was also the smallest. We then further extended the compressed output of RLZ to construct a self-index. Various data structures were explored to efficiently support the count() and locate() queries, and the algorithms for implementing these search queries were also discussed. However, these search queries are unable to detect occurrences of patterns spanning more than two factors, which is a limitation for its practical use. We also presented experimental results of the RLZ search query performance and compared it to the equivalent queries implemented by the RLCSA and LZ-End indexes. Unfortunately, the RLZ search query performance was very slow compared to these existing implementations. However, we were able to construct an index that was of comparable size to an LZ-End index. Several suggestions for improving the query speeds were discussed for future research.

The RLZ algorithm introduced in Chapter 6 is a relative compression algorithm. For most of the collections we used to test the performance of RLZ, a reference genome for the species was available. However, certain collections may not have a reference genome, or another sequence that is not the reference genome may better represent the sequences in the collection to produce better compression results. We discussed such issues in Chapter 8. First we showed the variation in the compressed collection size that can occur as the reference is varied. This showed that choosing an inappropriate reference could lead to compressed sizes that are several megabytes larger than when the best-available reference is used. To achieve good compression, the reference must represent the repetitions in the collections. Therefore, we experimented with constructing artificial references. We compressed our test collections with Re-Pair, COMRAD, dna-x and Sequitur, then constructed references using the repeats detected by these dictionary compression tools. The collections were then compressed with RLZ using the artificial references. The experimental results showed that using references constructed in this manner led to the compressed sizes being halved compared to using optimised RLZ for certain collections, especially when using a reference constructed with Re-pair. Unfortunately using two compression
algorithms to compress a collection cannot scale for large collections as the compression speed is slow and memory use of tools such as \texttt{Re-pair} are high. Therefore, we used several heuristics to detect repeats in a collection to construct a reference more efficiently. The most effective method we discovered was using supermaximal repeats. This technique will be explored further in future research.

In the next section, we discuss some future extensions for the \texttt{COMRAD} and \texttt{RLZ} algorithms, and also discuss some potential research questions to be explored.

**Future research**

The aim of our research project was to provide researchers that handle large collections of DNA data a set of tools to manage these collections. Our algorithms are practical for compressing the output of sequencing projects, especially our \texttt{RLZ} algorithm, which also has an improved variant \texttt{GDC} [Deorowicz and Grabowski, 2011]. Although researchers in the field have emphasised the importance of algorithms for handling large DNA datasets [Wooley \textit{et al.}, 2010], the research community is stagnant in adopting these technologies and to this day, continue to use tools such as \texttt{gzip} and \texttt{bzip2} for compression. This is unsurprising as these tools are widely available across many platforms and are regularly maintained, unlike DNA compression algorithms. Given that several practical algorithms are available, the focus should be on engineering these algorithms to build robust implementations that are distributed in software repositories so that they can be adopted by researchers.

Another focus of this research field should be on integrating self-indexes with DNA sequence analysis tools. As the volume of data to be analysed grows, it is difficult to manage it in an uncompressed form, so the ability to analyse the data in its compressed form will be desirable. Self-indexes allow basic queries to be invoked on compressed data, such as extracting substrings, and counting and returning occurrences of patterns [Arroyuelo \textit{et al.}, 2012]. Other complex queries such as searching for approximately matching strings can be built on the basic queries. Tools that analyse DNA or other types of data should interface with self-indexes that implement the necessary queries to conduct the data analysis, so that the data can be stored efficiently. In this manner, self-indexes can act as space-efficient databases.

On a related note, self-indexes tailored for specific applications may also be necessary. Most current indexes are designed for storage of strings [Kreft and Navarro, 2011; Mäkinen \textit{et al.}, 2010]. More specialised self-indexes also exist, such as for storing and searching exon data [Thachuk, 2011]. An example of a specialised self-index that should be explored is one for read data, which stores the reads, read identifiers, and quality scores in a compressed form, and is able to interface with assembly tools. Another example is a self-index for variation data, such as those from the 1000 Genomes project, so that researchers can query the self-index for genomes that contain a particular SNP or other type of mutation.
In terms of the algorithms presented in this thesis, we discussed various improvements that should be made to various aspects of **COMRAD** in Sections 5.2.7 and 5.3.4, **RLZ** in Sections 6.3, 7.1.6 and 7.2.5, and reference construction in Section 8.2.2. Overall, **COMRAD** should be extended to a fully disk-based algorithm to enable compression of large collections ranging from terabases to petabases in size. To achieve this, significant modifications to the substring frequency counting step and the substitution step are required. Since a fully disk-based algorithm is likely to be slow to compress and decompress, the algorithm must be extended to add or delete sequences from compressed collections, and enable certain other queries to be executed on compressed collections. An algorithm of this type may be useful for storing and managing databases like GenBank. Similarly for **RLZ**, the algorithm needs to be made more efficient for compressing much larger collections. However, as a priority, the focus should be on making the **RLZ** self-index search query more practical.

We also briefly explored several applications for the **COMRAD** and **RLZ** compressed output during this research project that were not discussed in this thesis, but they are applications that should be explored more concretely in future research. One such application is to conduct pair-wise comparisons of sequences in a collection using the non-terminals generated by **COMRAD** or the factors generated by **RLZ**. Two sequences that share non-terminals or factors are likely to be similar. Our initial experiments showed that the sequences in our **Mito** test collection could be clustered according to the genus of the species using the similarity of **RLZ** factors. This type of similarity metric could be used for sequence clustering or sequence alignment, and may be more efficient than using existing clustering or alignment algorithms.

Another such application is the use of multiple references to compress multi-species collections based on the phylogenetic relationship between species. As an example, given a collection containing five genomes each from the yeast species *S. cerevisiae*, *S. paradoxus* and *S. castelli*, the genomes from each species will be compressed separately with respect to a genome from the species. Then the reference *S. cerevisiae* genome will be compressed with respect to the reference *S. paradoxus* genome, and the reference *S. paradoxus* genome in turn will be compressed with respect to the reference *S. castelli* genome. The compression ordering is based on the hierarchy of the phylogenetic relationship between the species. Since phylogenetic relationships may not be available, heuristics are required to determine this hierarchy. There are also issues associated with maintaining multiple references that need to be explored, especially if the data is to be stored as a self-index.

This concludes our summary of the thesis and the discussion of future research in this area. As shown in this thesis, there are many improvements that are yet to be made to efficiently manage the large quantities of data output by sequencing projects. With sufficient extensions, such as parallelisation of the algorithms, the algorithms we presented in this thesis can form the basis for future algorithms that manage the rapidly increasing quantities of biomedical data.
Bibliography


Appendix A

Data and algorithm sources

Dataset sources

Sources for sequences in DATASET-SIN:

- *P. falciparum* genome
  plasmodb.org/common/downloads/release-7.1
  Accessed: 11 Jan 2011

- *C. elegans* genome
  www.sanger.ac.uk/Projects/C_elegans/Genomic_Sequence.shtml
  Accessed: 10 Jun 2010

- *A. thaliana* genome
  Accessed: 27 Jul 2010

- *E. coli* genome
  Accessed: 9 Aug 2010

- *D. melanogaster* genome
  Accessed: 07 Dec 2009

- *G. gallus* genome
  Accessed: 20 Feb 2009

- *S. cerevisiae* genome
  Accessed: 13 Feb 2009
• *H. sapiens* reference genome GRCh 37  
  Accessed: 1 Mar 2010

Sources for collections in DATASET-REP:

• **Hemoglobin** sequences  
  Search query at srs.ebi.ac.uk with description field matching *Hemoglobin*  
  Accessed: 23 Mar 2009

• **Influenza** sequences  
  Accessed: 2008

• **Mitochondria** genomes  
  Accessed: 27 Apr 2009

• **A. thaliana** contigs from 1001 Genomes Project  
  1001genomes.org/data/mpi/MPISchneeberger2011/releases/2010_10_04/Strains  
  Accessed: 11 Jan 2011  
  These sequence data were produced by the Weigel laboratory at the Max Planck Institute for Developmental Biology.

• **S. cerevisiae** genomes  
  Accessed: 13 Feb 2009

• **S. paradoxus** genomes  
  ftp://ftp.sanger.ac.uk/pub/dmc/yeast/latest/para_assemblies.tgz  
  Accessed: 13 Feb 2009

• **E. coli** genomes  
  Accessed: 9 Aug 2010

• **S. sonnei** genomes  
  Personal correspondence with Dr Kathryn Holt  
  Accessed: 26 May 2011

• **S. sonnei** reference genome  
  Accessed: 1 Jun 2011
• *H. sapiens* reference genome GRCh 37  
All files matching hs_ref_GRC37_chr*.fa.gz  
Accessed: 1 Mar 2010

• *Craig Venter* genome  
All files matching hs_alt_HuRef_chr*.fa.gz  
Accessed: 1 Mar 2010

• *Han Chinese* genome  
ftp://public.genomics.org.cn/BGI/yanhuang/fa  
Accessed: 19 Feb 2010

• *Korean* genome  
Accessed: 12 May 2010

• *Bacterial* genomes  
Accessed: 17 Mar 2009

**Algorithm and library sources**

**General-purpose compression algorithm implementations**

• gzip Version 1.3.12  
  packages.ubuntu.com/hardy/gzip

• 7-Zip Version 4.57  
  packages.ubuntu.com/hardy/p7zip-full

• compress Version 4.2.4.4  
  sourceforge.net/projects/ncompress/files

• Sequitur  
  code.google.com/p/sequitur/source/browse

• Re-pair Version 1.0.1  
  www.cbrc.jp/~rwan/en/restore.html

• ppmd Version 9.1-13  
  packages.ubuntu.com/hardy/ppmd

• bzip2 Version 1.0.4  
  packages.ubuntu.com/hardy/bzip2
APPENDIX A. DATA AND ALGORITHM SOURCES

DNA compression algorithm implementations

- **dna-x** Version 0.1.0
  Obtained through personal communication with the authors

- **GenCompress** updated version as of 18/01/2008
  www1.spms.ntu.edu.sg/~chenxin/GenCompress

- **XMCompress** Version 2.2
  Obtained through personal communication with the authors

- **DNACompress**
  monod.uwaterloo.ca/downloads/dnacompress/DNACompress.zip

- **Biocompress**
  hal.inria.fr/inria-00180949/fr

- **gdc** Version 0.3
  sun.aei.polsl.pl/gdc/download.html

Self-index implementations

- **RLCSA** version as at 25/11/2009
  www.cs.helsinki.fi/group/suds/rlcsa

- **LZ-End** version as at 07/02/2011
  pizzachili.dcc.uchile.cl/indexes/LZ77-index

Miscellaneous

- **libdivsufsort** Version 2.0.1
  code.google.com/p/libdivsufsort

- **lcpdc**
  goanna.cs.rmit.edu.au/~sjp/lcpdc.tar.gz

- **libcdds** Version 1.0.8
  libcdds.recoded.cl

- **RepeatMasker** Open-3.0 1996–2010
  www.repeatmasker.org

- **RepeatScout** Version 1.0.5
  bix.ucsd.edu/repeatscout

- **RECON** Version 1.05
  selab.janelia.org/recon.html
Appendix B

COMRAD implementation

In this section we describe the implementation details of COMRAD. COMRAD aims to compress collections in the gigabase to terabase range, and detects repeats globally. Therefore, it is important to pay careful attention to the implementation, especially the frequency dictionary construction step of each iteration, which may need to store many distinct substrings and their frequencies. We discuss the important aspects of the implementation details of each step of the algorithm.

**Input:** The input into the algorithm is expected to be one or more files of DNA sequences, and each file must be in the FASTA format, which was described in Section 2.1.5. Each file can contain one or more FASTA entries. Each entry will be considered to be a separate sequence. For example, if the input consists of 5 files, where each file contains 10 FASTA entries, then the input collection $S_0$ contains 50 sequences, where the sequences correspond to the FASTA entries in the files.

**Frequency dictionary construction for first iteration:** During our experiments, we found that the dictionary construction step of the first iteration is the most memory intensive step of the algorithm. This is especially the case for non-repetitive sequences containing many distinct $L$-mers, for reasonably large values of $L$ ($L \geq 15$). We explored two variations for storing frequencies of distinct $L$-mers.

**Approach 1:** The first approach we adopted was to store the frequency dictionary as a hash table. Only the counts of distinct $L$-mers occurring in the sequences are stored, reducing the space consumption for repetitive sequences with few distinct $L$-mers. An entry can be searched for, added or updated in amortised $O(1)$ time using a hash table, making it ideal for frequency counting. The main disadvantage is the memory required by the extra slots necessary to maintain the $O(1)$ access and update time. Even with the extra memory use, we found that for small datasets (less than 1 Gbase), using a hash table was more memory efficient than the other alternative, which we explain shortly.

We chose the value $L = 16$ to be consistent with our aim of choosing an initial
substring length that is long enough so that the chance of detecting a randomly occurring L-mer as a repeat is low, but short enough to capture any longer repeats. We stored an L-mer in a 64-bit integer, using 4 bits to represent a nucleotide. This representation requires $N_1^{DL}(8 + 4) = 12N_1^{DL}$ bytes for storing $N_1^{DL}$ distinct L-mers (8 bytes for storing the L-mer and 4 bytes for storing the frequency). We attempt to keep the hash table approximately 60% full to reduce the possibility of collisions. Therefore, the space used by the hash table is approximately $20N_1^{DL}$ bytes. In practice, we use 2 bpb to store L-mers containing only nucleotides from alphabet $\sigma$, and 4 bpb to store L-mers containing at least one nucleotide from alphabet $\Sigma$. Most L-mers tend to only contain nucleotides from $\sigma$ so the space use is closer to $\frac{40}{3}N_1^{DL}$.

If an input collection is larger than 1 Gbase, then the memory use with $L = 16$ can be significant. For a 1 Gbase non-repetitive collection, it is possible that the collection contains $10^9$ distinct L-mers. This requires $20 \times 10^9$ bytes of space. In general, for $L = 16$, there are $N_1^{DL} = 4^{16} \approx 4$ billion distinct L-mers from the $\sigma$ alphabet, and many more from the $\Sigma$ alphabet. This requires approximately 82 Gbyte of space. For large non-repetitive collections, the hash table approach to constructing the frequency dictionary is infeasible, so an alternative is required.

**Approach 2:** The space cost of the previous solution consisted of the space required to store the L-mer and its frequency (12 bytes per hash table entry). If we expect that most or all distinct L-mers will appear in a collection, then only the space required to count the frequency of all distinct L-mers can be allocated. Since this is infeasible for the $\Sigma$ alphabet, we restrict ourselves to just the $\sigma$ alphabet. For $L = 16$, there are $2^{32}$ distinct L-mers. Since using 4 bytes to store each count requires $2^{34}$ bytes, we restrict the frequencies to a maximum of 255 to use just one byte per count. The space use is now restricted to just $2^{32}$ bytes. The frequencies do not need to be accurate, and 255 is a sufficiently large frequency to distinguish between frequent and infrequent L-mers.

So far we have ignored the frequencies of L-mers containing nucleotides from the $\Sigma$ alphabet. Since L-mers containing nucleotides from $\Sigma$ are rare and unlikely to be frequent, we discard the counts of all such L-mers except for the count for the L-mer of all N nucleotides. Substrings of Ns are used typically in assembled sequences to represent gaps and other regions that could not be sequenced accurately, therefore it is likely to occur frequently and should not be ignored.

The substitution step also requires non-terminal identifiers associated with each frequent L-mer, and this requires 4 bytes per L-mer. For $L = 15$ and $L = 16$, the memory use is then $2^{32}$ and $2^{34}$ bytes, respectively, which is a significant amount of memory. Therefore, this approach will only be used for compressing collections for which the hash table requires more memory than is available in our test system. For small, highly-repetitive collections, using the first approach is more memory efficient. As a rule of thumb, we use the first approach for compressing collections of length less than 1 Gbase and use the second approach for larger collections.
Frequency dictionary construction for subsequent iterations: The space usage for this step depends on the repeat properties of the collection being compressed. It is difficult to know the number of distinct substrings that will be added to the frequency dictionary so, once again, we use a hash table. Recall that a substring at this step consists of both terminals and non-terminals. Therefore, we store a substring as an integer array, where the range of negative integers from -1 to -15 represent nucleotides, a 1 represents a reverse complement symbol, and the remaining positive integers represent non-terminals. Since the longest substring permitted by our set of patterns is of length 6, and at most two extra symbols may be required if there are two reverse complement non-terminals, a substring can use at most 32 bytes. Another 4 bytes are required to store the frequency. Therefore, a hash table entry requires 36 bytes. If there are $N_{DP}^k$ distinct pattern substrings at the $k$th iteration, then the hash table can occupy approximately $60N_{DP}^k$ bytes of memory (assuming a 60% full hash table).

The space usage can be significant if the number of non-terminals created in the previous ($k-1$) iterations is high. To limit the memory usage, we use the following observation. If a substring matching a pattern in $P$ found in the $k$th iteration contains no non-terminals introduced in the ($k-1$)th iteration, then this substring is guaranteed to have been considered for replacement in an iteration before the ($k-1$)th iteration. Given that the substring was not replaced earlier, it is not frequent, so should not be added to the frequency dictionary at the $k$th iteration. This restriction limits the number of distinct substrings $N_{DP}^k$ that can be added to the hash table at a given iteration, to a factor of the number of non-terminals introduced in the previous iteration. In other words, $N_{DP}^k$ is proportional to $|R_{k-1}|$.

It is possible that a large number of non-terminals are introduced in the previous iteration, resulting in many distinct substrings that cannot fit into memory. During any of the iterations, we cannot guarantee that the frequency dictionary will fit in memory. The algorithm does not take memory limits as an input to the program, and it does not limit the number of substrings that will be added to the dictionary. Currently, for compressing a relatively non-repetitive collection with a few gigabases of nucleotides requires around 20-30 Gbytes of memory. It should be noted that limiting memory use and the number of substrings added to the dictionary can reduce the compression performance. Some of these potential improvements are discussed in Section 5.2.7.

Substitution step: The frequency dictionary for the current iteration needs to be in memory during the substitution step. By this stage, the dictionary only contains the frequently occurring substrings, therefore, the space used by the dictionary is much less than the quantities reported above.

To make the substitutions, a single sequence at a time is stored in memory. In the first iteration, we use a byte array to store the sequence, and an array of in-
tegers to store the frequencies of each \( L \)-mer in the sequence, which will later be sorted to determine the series of substitutions. A bit array is also stored to indicate which substrings are forward substitutions and which are reverse complement substitutions. Using this representation, each sequence \( S_0 \) can be stored using approximately \( 5n_0 + n_0/8 \) bytes. In subsequent iterations, sequences contain both nucleotides and non-terminals. Therefore, we store the sequence using the integer representation used to store distinct pattern substrings in the hash table. The array of integers containing substring frequencies and the bit array that identifies reverse complement substitutions, are also stored. Using this representation, each sequence \( S_{k-1} \) can be stored using approximately \( 8n_{k-1} + n_{k-1}/8 \) bytes. To sort the substring frequencies, we used the \texttt{qsor}t quick-sort implementation in C.

If a sequence is too large to fit in memory (tens of gigabases or more), then the sequence can be split into multiple smaller sequences without reducing the compression performance. Care should be taken to preserve any repeats that occur across the boundaries where the sequences are split.

**Dictionary cleanup:** The dictionary cleanup step requires the substitution rules from all the iterations to be available in memory to maintain an accurate count of the non-terminals as replacements are made. Also the substrings represented by infrequent non-terminals can be accessed quickly for replacement. We use the same techniques as in the first and subsequent iteration dictionary representations to store the substitution rules, which can be memory intensive if there are many substitution rules. In our experiments, we observed that for most collection, the memory used during this step is much less than the memory used in the first iteration dictionary construction step. However, we have encountered large collections of over 100 Gbases with more substitution rules than can be stored in a 32 Gbytes of RAM. Disk-based solutions to the dictionary cleanup step will be explored in future work, as it will be necessary for compressing terabase sized collections. Although the cleanup step is not necessary, it will improve the compression result.

**Decompression:** The Huffman coding and decoding implementations used are standard, so we do not elaborate on these steps. For fast decompression of \texttt{COMRAD}-compressed sequences, substrings represented by each non-terminal must be in memory. This may be expensive if a large number of non-terminals represent long repeats. Currently, we use \( x \) bytes for a non-terminal whose substring contains \( x \) nucleotides.
Author/s: Kuruppu, Shanika Sewwandini

Title: Compression of large DNA databases

Date: 2012


Persistent Link: http://hdl.handle.net/11343/37308

File Description: Compression of large DNA databases

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.