Goldsmiths
UNIVERSITY
OF LONDON

BSc in
Computing and Information Systems

Data compression

I. Pu

2004

2910325

UNIVERSITY OF LONDON
18 58
EXTERNAL PROGRAMME

This guide was prepared for the University of London by:

I. Pu, PhD, BSc, Lecturer, Department of Computing, Goldsmiths College, University of London.

This is one of a series of subject guides published by the University. We regret that due to pressure of work the authors are unable to enter into any correspondence relating to, or arising from, the guide. If you have any comments on this subject guide, favourable or unfavourable, please use the form at the back of this guide.

The External Programme
Publications Office
University of London
Stewart House
32 Russell Square
London WC1B 5DN
United Kingdom

Web site: www.londonexternal.ac.uk

Published by: University of London Press

Printed by: Central Printing Service, University of London, England

Contents

Chapter 0

Introduction

Data compression

Data compression is the science (and art) of representing information in a compact form. Having been the domain of a relatively small group of engineers and scientists, it is now ubiquitous. It has been one of the critical enabling technologies for the on-going digital multimedia revolution for decades. Without compression techniques, none of the ever-growing Internet, digital TV, mobile communication or increasing video communication would have been practical developments.

Data compression is an active research area in computer science. By 'compressing data', we actually mean deriving techniques or, more specifically, designing efficient algorithms to:

- represent data in a less redundant fashion
- remove the redundancy in data
- implement coding, including both encoding and decoding.

The key approaches of data compression can be summarised as *modelling + coding*. Modelling is a process of constructing a knowledge system for performing compression. Coding includes the design of the code and product of the compact data form.

Aims and objectives of the subject

The subject aims to introduce you to the main issues in data compression and common compression techniques for text, audio, image and video data and to show you the significance of some compression technologies.

The objectives of the subject are to:

- outline important issues in data compression
- describe a variety of data compression techniques
- explain the techniques for compression of binary programmes, data, sound and image
- describe elementary techniques for modelling data and the issues relating to modelling.

Motivation for studying the subject

You will broaden knowledge of compression techniques as well as the mathematical foundations of data compression, become aware of existing compression standards and some compression utilities available. You will also benefit from the development of transferable skills such as problem analysis and problem solving. You can also improve your programming skills by doing the laboratory work for this subject.

Textbooks

There are a limited number of books on Data compression available. No single book is completely satisfactory to be used as the textbook for the subject. Therefore, instead of recommending one book for essential reading and a few books for further reading, we recommend chapters of some books for essential reading and chapters from some other books for further reading at the beginning of each chapter of this subject guide. An additional list of the books recommended for support and for historical background reading is attached in the reading list (Appendix E).

Reading

Salomon, David *A Guide to Data Compression Methods.* (London: Springer, 2001) [ISBN 0-387-95260-8].

Wayner, Peter *Compression Algorithms for Real Programmers.* (London: Morgan Kaufmann, 2000) [ISBN 0-12-788774-1].

Chapman, Nigel and Chapman, Jenny *Digital Multimedia.* (Chichester: John Wiley & Sons, 2000) [ISBN 0-471-98386-1].

Sayood, Khalid *Introduction to Data Compression.* 2nd edition (San Diego: Morgan Kaufmann, 2000) [ISBN 1-55860-558-4].

Web addresses

`www.datacompression.com`

Prerequisites

The prerequisites for this subject include a knowledge of elementary mathematics and basic algorithmics. You should review the main topics in mathematics, such as sets, basic probability theory, basic computation on matrices and simple trigonometric functions (e.g. sin(x) and cos(x)), and topics in algorithms, such as data structures, storage and efficiency. Familiarity with the elements of computer systems and networks is also desirable.

Study methods

As experts have predicted that more and more people in future will apply computing for multimedia, we recommend that you learn the important principles and pay attention to understanding the issues in the field of data compression. The experience could be very useful for your future career.

We suggest and recommend highly the following specifically:

1. Spend two hours on revision or exercise for every hour of study on new material.
2. Use examples to increase your understanding of new concepts, issues and problems.
3. Always ask the question: ‘Is there a better solution for the current problem?’
4. Use the Content pages to view the scope of the subject; use the Learning Outcomes at the end of each chapter and the Index pages for revision.

Exercises, programming laboratory and courseworks

It is useful to have access to a computer so that you can actually implement the algorithms learnt for the subject. There is no restriction on the computer platform nor requirement of a specific procedural computer language. Examples of languages recommended include Java, C, C++ or even Pascal.

Courseworks (issued separately every year) and tutorial or exercise/lab sheets (see Activities section and Sample examination questions at the end of each chapter) are set for you to check your understanding or to practice your programming skills using the theoretical knowledge gained from the course.

The approach to implementing an algorithm can be different when done by different people, but it generally includes the following stages of work:

1. Analyse and understand the algorithm
2. Derive a general plan for implementation
3. Develop the program
4. Test the correctness of the program
5. Comment on the limitations of the program.

At the end, a full document should be written which includes a section for each of the above stages of work.

Examination

The content in this subject guide will be examined in a two-hour-15-minute examination[1]. At the end of each chapter, there are sample examination questions for you to work on.

[1] *See the sample examination papers in Appendix A,C and solutions in Appendix B,D*

You will normally be required to answer three out of five or four out of six questions. Each question often contains several subsections. These subsections may be classified as one of the following three types:

- **Bookwork** The answers to these questions can be found in the subject guide or in the main textbook.
- **Similar question** The questions are similar to an example in the subject guide or the main textbook.
- **Unseen question** You may have not seen these types of questions before but you should be able to answer them using the knowledge and experience gained from the subject.

More information on how to prepare for your examination can be found in Chapter 13.

Subject guide

The subject guide covers the main topics in the syllabus. It can be used as a reference which summarises, highlights and draws attention to some important points of the subject. The topics in the subject guide are equivalent to the material covered in a one term third-year module of BSc course in Mathematics, Computer Science, Internet Computing, or Computer Information Systems in London, which totals thirty-three hours of lectures, ten hours of supervised laboratory work, and twenty hours of recommended individual revisions or implementation. The subject guide is for those students who have completed all second year courses and have a successful experience of programming.

This subject guide sets out a sequence to enable you to efficiently study the topics covered in the subject. It provides guidance for further reading, particularly in those areas which are not covered adequately in the course.

It is unnecessary to read every textbook recommended in the subject guide. One or two books should be enough to enable individual topics to be studied in depth. One effective way to study the compression algorithms in the module is to trace the steps in each algorithm and attempt an example by yourself. Exercises and courseworks are good opportunities to help understanding. The sample examination paper at the end of the subject guide may also provide useful information about the type of questions you might expect in the examination.

One thing the reader should always bear in mind is the fact that Data compression, like any other active research area in Computer Science, has kept evolving and has been updated, sometimes at an annoyingly rapid pace. Some of the descriptive information provided in any text will eventually become outdated. Hence you should try not to be surprised if you find different approaches, explanations or results among the books you read including this subject guide. The learning process requires the **input** of your own experiments and experience. Therefore, you are encouraged to, if possible, pursue articles in research journals, browse the relative web sites, read the latest versions of books, attend conferences or trade shows etc., and in general pay attention to what is happening in the computing world.

The contents of this subject guide are arranged as follows: *Chapter 1* discusses essentials of Data compression, including a very brief history. *Chapter 2* introduces an intuitive compression method: Run-length coding. *Chapter 3* discusses the preliminaries of data compression, reviews the main idea of Huffman coding, and Shannon-Fano coding. *Chapter 4* introduces the concepts of prefix codes. *Chapter 5* discusses Huffman coding again, applying the information theory learnt, and derives an efficient implementation of Huffman coding. *Chapter 6* introduces adaptive Huffman coding. *Chapter 7* studies issues of Arithmetic coding. *Chapter 8* covers dictionary-based compression techniques. *Chapter 9* discusses image data and explains related issues. *Chapter 10* considers image compression techniques. *Chapter 11* introduces video compression methods. *Chapter 12* covers audio compression, and finally, *Chapter 13* provides information on revision and examination. At the end of each chapter, there are Learning outcomes, Activities, laboratory questions and selected Sample examination questions. At the end of the subject guide, two sample examination papers and solutions from previous examinations in London can be found in the Appendix A-D.

Activities

1. Review your knowledge of one high level programming language of your choice, e.g. Java or C.

2. Review the following topics from your earlier studies of elementary mathematics and basic algorithmics:

   - sets
   - basic probability theory
   - basic computation on matrices
   - basic trigonometric functions
   - data structures, storage and efficiency.
   - the elements of computer systems and networks

Laboratory

1. Design and implement a programme in Java (or in C, C++) which displays a set of English letters occurred in a given string (upper case only).

   For example, if the user types in a string "AAABBEECEDE", your programme should display "(A,B,E,C,D)".

   The user interface should be something like this:

   ```
   Please input a string:
   > AAABBEECEDE
   The letter set is:
   (A,B,E,C,D)
   ```

2. Write a method that takes a string (upper case only) as a parameter and that returns a histogram of the letters in the string. The ith element of the histogram should contain the number of ith character in the string alphabet.

   For example, if the user types in a string "AAABBEECEDEDEDDDE", then the string alphabet is "(A,B,E,C,D)". The output could be something like this:

   ```
   Please input a string:
   > AAABBEECEDEDEDDDE
   The histogram is:
   A xxx
   B xx
   E xxxxxx
   C x
   D xxxxx
   ```

Chapter 1

Data compression

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 1.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Introduction.

Importance of data compression

Data compression techniques is motivated mainly by the need to improve efficiency of information processing. This includes improving the following main aspects in the digital domain:

- storage efficiency
- efficient usage of transmission bandwidth
- reduction of transmission time.

Although the cost of storage and transmission bandwidth for digital data have dropped dramatically, the demand for increasing their capacity in many applications has been growing rapidly ever since. There are cases in which extra storage or extra bandwidth is difficult to achieve, if not impossible. Data compression as a means may make much more efficient use of existing resources with less cost. Active research on data compression can lead to innovative new products and help provide better services.

Brief history

Data compression can be viewed as the art of creating shorthand representations for the data even today, but this process started as early as 1,000 BC. The short list below gives a brief survey of the historical milestones:

- 1000BC, Shorthand
- 1829, Braille code
- 1843, Morse code
- 1930 onwards, Analog compression
- 1950, Huffman codes
- 1975, Arithmetic coding
- 1977, Dictionary-based compression
- 1980s
  - early 80s, FAX
  - mid-80s, Video conferencing, still images (JPEG), improved FAX standard (JBIG)
  - late 80s, onward Motion video compression (MPEG)
- 1990s
  - early 90s, Disk compression (stacker)
  - mid-90s, Satellite TV
  - late 90s, Digital TV (HDTV), DVD, MP3

- 2000s Digital TV (HDTV), DVD, MP3

Source data

In this subject guide, the word *data* includes any digital information that can be processed in a computer, which includes text, voice, video, still images, audio and movies. The data before any compression (i.e. encoding) process is called the *source data*, or the *source* for short.

Three common types of source data in the computer are *text* and (digital) *image* and *sound*.

- **Text** data is usually represented by ASCII code (or EBCDIC).
- **Image** data is represented often by a two-dimensional array of *pixels* in which each pixel is associated with its color code.
- **Sound** data is represented by a wave (periodic) function.

In the application world, the source data to be compressed is likely to be so-called *multimedia* and can be a mixture of text, image and sound.

Lossless and lossy data compression

Data compression is simply a means for efficient digital representation of a source of data such as text, image and the sound. The goal of data compression is to represent a source in digital form with as few bits as possible while meeting the minimum requirement of reconstruction. This goal is achieved by removing any redundancy presented in the source.

There are two major families of compression techniques in terms of the possibility of reconstructing the original source. They are called *Lossless* and *lossy* compression.

Lossless compression

A compression approach is lossless only if it is possible to exactly reconstruct the original data from the compressed version. There is no loss of any information during the compression[1] process.

[1] *This, when used as a general term, actually includes both compression and decompression process.*

Lossless compression techniques are mostly applied to symbolic data such as character text, numeric data, computer source code and executable graphics and icons.

Lossless compression techniques are also used when the original data of a source are so important that we cannot afford to lose any details. For example, medical images, text and images preserved for legal reasons; some computer executable files, etc.

Lossy compression

A compression method is lossy compression only if it is not possible to reconstruct the original exactly from the compressed version. There are some insignificant details that may get lost during the process of compression.

Approximate reconstruction may be very good in terms of the compression-ratio but usually it often requires a trade-off between the visual quality and the computation complexity (i.e. speed).

Data such as multimedia images, video and audio are more easily compressed by lossy compression techniques.

Main compression techniques

Data compression is often called *coding* due to the fact that its aim is to find a specific *short* (or shorter) way of representing data. *Encoding* and *decoding* are used to mean compression and decompression respectively. We outline some major compression algorithms below:

- Run-length coding
- Quantisation
- Statistical coding
- Dictionary-based coding
- Transform-based coding
- Motion prediction.

Run-length coding

The idea of Run-length coding is to replace consecutively repeated symbols in a source with a code pair which consists of either the repeating symbol and the number of its occurrences, or sequence of non-repeating symbols.

Example 1.1 *String* `ABBBBBBBCC` *can be represented by* $\texttt{Ar}_7\texttt{Br}_2\texttt{C}$*, where* r_7 *and* r_2 *means 7 and 2 occurrences respectively.*

All the symbols are represented by an 8-bit ASCII codeword.

Quantisation

The basic idea of quantisation is to apply a certain computation to a set of data in order to achieve an approximation in a simpler form.

Example 1.2 *Consider storing a set of integers (7, 223, 15, 28, 64, 37, 145). Let* x *be an integer in the set. We have* $7 \leq x \leq 223$*. Since* $0 < x < 255$ *and* $2^8 = 256$*, it needs 8 binary bits to represent each integer above.*

However, if we use a multiple, say 16, as a common divider to apply to each integer and round its value to the nearest integer, the above set becomes (0, 14, 1, 2, 4, 2, 9) after applying the computation x div 16. Now each integer can be stored in 4 bits, since the maximum number 14 is less than $2^4 = 16$.

Statistical coding

The idea of statistical coding is to use statistical information to replace a fixed-size code of symbols by a, hopefully, shorter variable-sized code.

Example 1.3 *We can code the more frequently occurring symbols with fewer bits. The statistical information can be obtained by simply counting the frequency of each character in a file. Alternatively, we can simply use the probability of each character.*

Dictionary-based coding

The dictionary approach consists of the following main steps:

1. read the file
2. find the frequently occurring sequences of symbols (FOSSs)
3. build up a dictionary of these FOSSs
4. associate each sequence with an index (usually a fixed length code)
5. replace the FOSS occurrences with the indices.

Transform-based coding

The transform-based approach models data by mathematical functions, usually by periodic functions such as $cos(x)$ and applies mathematical rules to primarily diffuse data. The idea is to change a mathematical quantity such as a sequence of numbers[2] to another form with useful features. It is used mainly in lossy compression algorithms involving the following activities:

[2]*... or anything else*

- analysing the signal (sound, picture etc.)
- decomposing it into frequency components
- making use of the limitations of human perception.

Motion prediction

Again, motion prediction techniques are lossy compression for sound and moving images.

Here we replace objects (say, an 8×8 block of pixels) in frames with references to the same object (at a slightly different position) in the previous frame.

Compression problems

In this course, we view data compression as algorithmic problems. We are mainly interested in compression algorithms for various types of data.

There are two classes of compression problems of interest (Davisson and Gray 1976):

- **Distortion-rate problem** Given a constraint on transmitted data rate or storage capacity, the problem is to compress the source at, or below, this rate but at the highest fidelity possible.
  Compression in areas of voice mail, digital cellular mobile radio and video conferencing are examples of the distortion-rate problems.
- **Rate-distortion problem** Given the requirement to achieve a certain pre-specified fidelity, the problem is to meet the requirements with as few bits per second as possible.
  Compression in areas of CD-quality audio and motion-picture-quality video are examples of rate-distortion problems.

Algorithmic solutions

In areas of data compression studies, we essentially need to analyse the characteristics of the data to be compressed and hope to deduce some patterns in order to achieve a compact representation. This gives rise to a variety of data modelling and representation techniques, which are at the heart of compression techniques. Therefore, there is no 'one size fits all' solution for data compression problems.

Compression and decompression

Due to the nature of data compression, any compression algorithm will not work unless a decompression approach is also provided. We may use the term *compression algorithm* to actually mean both compression algorithm and the decompression algorithm. In this subject, we sometimes do not discuss the decompression algorithm when the decompression process is obvious or can be easily derived from the compression process. However, you should always make sure that you know the decompression solutions.

In many cases, the efficiency of the decompression algorithm is of more concern than that of the compression algorithm. For example, movies, photos, and audio data are often compressed once by the artist and then decompressed many times by millions of viewers. However, the efficiency of compression is sometimes more important. For example, programs may record audio or video files directly to computer storage.

Compression performance

The performance of a compression algorithm can be measured by various criteria. It depends on what is our priority concern. In this subject guide, we are mainly concerned with the effect that a compression makes (i.e. the difference in size of the input file before the compression and the size of the output after the compression).

It is difficult to measure the performance of a compression algorithm in general because its compression behaviour depends much on whether the data contains the right patterns that the algorithm looks for.

The easiest way to measure the effect of a compression is to use the compression ratio.

The aim is to measure the effect of a compression by the shrinkage of the size of the source in comparison with the size of the compressed version.

There are several ways of measuring the compression effect:

- **Compression ratio**. This is simply the ratio of size.after.compression to size.before.compression or

$$\text{Compression ratio} = \frac{\text{size.after.compression}}{\text{size.before.compression}}$$

- **Compression factor**. This is the reverse of *compression ratio*.

$$\text{Compression factor} = \frac{\text{size.before.compression}}{\text{size.after.compression}}$$

- **Saving percentage**. This shows the shrinkage as a percentage.

$$\text{Saving percentage} = \frac{\text{size.before.compression} - \text{size.after.compression}}{\text{size.before.compression}}\%$$

Note: some books (e.g. Sayood(2000)) defines the compression ratio as our compression factor.

Example 1.4 *A source image file (pixels* 256×256*) with 65,536 bytes is compressed into a file with 16,384 bytes. The compression ratio is 1/4 and the compression factor is 4. The saving percentage is: 75%*

In addition, the following criteria are normally of concern to the programmers:

- **Overhead**. Overhead is some amount of extra data added into the compressed version of the data. The overhead can be large sometimes although it is often much smaller than the space saved by compression.
- **Efficiency** This can be adopted from well established algorithm analysis techniques. For example, we use the big-O notation for the time efficiency and the storage requirement. However, compression algorithms' behaviour can be very inconsistent but it is possible to use past empirical results.
- **Compression time** We normally consider the time for encoding and for decoding separately. In some applications, the decoding time is more important than encoding. In other applications, they are equally important.
- **Entropy**[3]. If the compression algorithm is based on statistical results, then entropy can be used to help make a useful judgement.

[3] *We shall introduce the concept of entropy later*

Limits on lossless compression

How far can we go with a lossless compression? What is the best compression we can achieve in a general case? The following two statements may slightly surprise you:

1. No algorithm can compress all (possible) files, even by one byte.
2. No algorithm can compress even 1% of all (possible) files even by one byte.

An informal reasoning for the above statements can be found below:

1. Consider the compression of a `big.file` by a lossless compression algorithm called cmpres. If statement 1 were not true, we could then effectively repeat the compression process to the source file.

   By 'effectively', we mean that the compression ratio is always < 1. This means that the size of the compressed file is reduced every time when running programme cmpres. So cmpres(cmpres(cmpres($\cdots$ cmpres(`big.file`)$\cdots$))), the output file after compression many times, would be of size 0.

   Now it would be impossible to losslessly reconstruct the original.

2. Compressing a file can be viewed as mapping the file to a different (hopefully shorter) file.

   Compressing a file of n bytes (in size) by at least 1 byte means mapping the file of n bytes to a file of $n-1$ bytes or fewer bytes. There are $(2^8)^n = 256^n$ files of n bytes and 256^{n-1} of $n-1$ bytes in total. This means that the proportion of the successful 1-to-1 mappings is only $256^{n-1}/256^n = 1/256$ which is less than 1%.

Learning outcomes

On completion of your studies in this chapter, you should be able to:

- outline the brief history of Data compression

- explain how to distinguish lossless data compression from lossy data compression
- outline the main compression approaches
- measure the effect and efficiency of a data compression algorithm
- explain the limits of lossless compression.

Activities

1. Investigate what compression software is available on your computer system.
2. Suppose that you have compressed a file myfile using a compression utility available on your computer. What is the name of the compressed file?
3. Use a compression facility on your computer system to compress a text file called myfile containing the following text:

   This is a test.

   Suppose you get a compressed file called myfile.gz after compression. How would you measure the size of myfile and of myfile.gz?
4. Suppose the size of myfile.gz is 20 KB while the original file myfile is of size 40 KB. Compute the compression ratio, compression factor and saving percentage.
5. A compression process is often said to be 'negative' if its compression ratio is greater than 1.

   Explain why negative compression is an inevitable consequence of a lossless compression.

Laboratory

1. If you have access to a computer using Unix or Linux operating system, can you use `compress` or `gzip` command to compress a file?
2. If you have access to a PC with Windows, can you use `WinZip` to compress a file?
3. How would you recover your original file from a compressed file?
4. Can you use `uncompress` or `gunzip` command to recover the original file?
5. Implement a method `compressionRatio` in Java which takes two integer arguments `sizeBeforeCompression` and `sizeAfterCompression` and returns the compression ratio. See Activity 4 for example.
6. Similarly, implement a method `savingPercentage` in Java which takes two integer arguments `sizeBeforeCompression` and `sizeAfterCompression` and returns the saving percentage.

Sample examination questions

1. Explain briefly the meanings of *lossless* compression and *lossy* compression. For each type of compression, give an example of an application, explaining why it is appropriate.
2. Explain why the following statements are considered to be true in describing the absolute limits on lossless compression.

   - No algorithm can compress all files, even by one byte.
   - No algorithm can compress even 1% of all files, even by one byte.

Chapter 2

Run-length algorithms

Essential reading

Sayood, Khalid *Introduction to Data Compression* (Morgan Kaufmann, 1996) [ISBN 1-55860-346-8]. Chapter 6.8.1.

Run-length coding ideas

A run-length algorithm assigns codewords to consecutive recurrent symbols (called runs) instead of coding individual symbols. The main idea is to replace a number of consecutive repeating symbols by a short codeword unit containing three parts: a single symbol, a run-length count and an interpreting indicator.

Example 2.1 *String* `KKKKKKKKK`*, containing 9 consecutive repeating* `K`*s, can be replaced by a short unit* `r9K` *consisting of the symbol* `r`*,* `9` *and* `K`*, where* `r` *represents 'repeating symbol',* `9` *means '9 times of occurrence' and* `K` *indicates that this should be interpreted as 'symbol* `K`*' (repeating 9 times).*

Run-length algorithms are very effective if the data source contains many runs of consecutive symbol. The symbols can be characters in a text file, *0*s or *1*s in a binary file or black-and-white pixels in an image.

Although simple, run-length algorithms have been used well in practice. For example, the so-called HDC (Hardware Data Compression) algorithm, used by tape drives connected to IBM computer systems, and a similar algorithm used in the IBM SNA (System Network Architecture) standard for data communications are still in use today.

We briefly introduce the HDC algorithm below.

Hardware data compression (HDC)

In this form of run-length coding, the coder replaces sequences of consecutive identical symbols with three elements:

1. a single symbol
2. a run-length count
3. an indicator that signifies how the symbol and count are to be interpreted.

A simple HDC algorithm

This uses only ASCII codes for:

1. the single symbols, and
2. a total of 123 control characters with a run-length count, including:
   - repeating control characters: $r_2, r_3, \cdots, r_{63}$, and

- non-repeating control characters: $n_1, n_2, \cdots, n_{63}$.

Each r_i, where $i = 2 \cdots 63$, is followed by either another control character or a symbol. If the following symbol is another control character, r_i (alone) signifies i repeating space characters (i.e. blanks). Otherwise, r_i signifies that the symbol immediately after it repeats i times.

Each n_i, where $i = 1 \cdots 63$ is followed by a sequence of i *non-repeating* symbols.

Applying the following 'rules', it is easy to understand the outline of the *encoding* and *decoding* run-length algorithms below:

Encoding

Repeat the following until the end of input file:

Read the source (e.g. the input text) symbols sequentially and

[1] *i.e. a sequence of symbols.*

1. if a string[1] of i ($i = 2 \cdots 63$) consecutive spaces is found, output a single control character r_i
2. if a string of i ($i = 3 \cdots 63$) consecutive symbols other than spaces is found, output two characters: r_i followed by the repeating symbol
3. otherwise, identify a longest string of $i = 1 \cdots 63$ non-repeating symbols, where there is no consecutive sequence of 2 spaces or of 3 other characters, and output the non-repeating control character n_i followed by the string.

Example 2.2 `GGG␣␣␣␣␣␣BCDEFG␣␣55GHJK␣LM777777777777`

can be compressed to $r_3\mathtt{G}r_6n_6\mathtt{BCDEFG}r_2n_9\mathtt{55}\sqcup\mathtt{LM}r_{12}\mathtt{7}$

Solution

1. The first 3 `G`s are read and encoded by $r_3\mathtt{G}$.
2. The next 6 spaces are found and encoded by r_6.
3. The non-repeating symbols `BCDEFG` are found and encoded by $n_6\mathtt{BCDEFG}$.
4. The next 2 spaces are found and encoded by r_2.
5. The next 9 non-repeating symbols are found and encoded by $n_9\mathtt{55GHJK}\sqcup\mathtt{LM}$.
6. The next 12 '7's are found and encoded by $r_{12}7$.

Therefore the encoded output is: $r_3\mathtt{G}r_6n_6\mathtt{BCDEFG}r_2n_9\mathtt{55}\sqcup\mathtt{LM}r_{12}\mathtt{7}$.

Decoding

The decoding process is similar to the encoding and can be outlined as follows:

Repeat the following until the end of input coded file:

Read the codeword sequence sequentially and

1. if a r_i is found, then check the next codeword.
   (a) if the codeword is a control character, output i spaces.
   (b) otherwise output i (ASCII codes of) repeating symbols.
2. otherwise, output the next i non-repeating symbols.

Observation

1. It is not difficult to observe from a few examples that the performance of the HDC algorithm (as far as the compression ratio concerns) is:
   - excellent[2] when the data contains many runs of consecutive symbols
   - poor when there are many segments of non-repeating symbols.

   Therefore, run-length algorithms are often used as a subroutine in other more sophisticated coding.

2. The decoding process of HDC is simpler than the encoding one[3]
3. The HDC is non-adaptive because the model remains unchanged during the coding process.

[2] *It can be even better than entropy coding such as Huffman coding.*

[3] *HDC is one of the so-called 'asymmetric' coding methods. Please see page 24 for definition.*

Learning outcomes

On completion of your studies in this chapter, you should be able to:

- state what a Run-length algorithm is
- explain how a Run-length algorithm works
- explain under what conditions a Run-length algorithm may work effectively
- explain, with an example, how the HDC algorithm works.

Activities

1. Apply the HDC (Hardware Data Compression) algorithm to the following sequence of symbols:

   `kkkk␣␣␣␣␣␣␣␣␣␣␣␣g␣␣hh5522777666abbbbcmmj␣␣##`

   Show the compressed output and explain the meaning of each control symbol.

2. Explain how the compressed output from the above question can be reconstructed using the decompressing algorithm.

3. Provide an example of a source file on which the HDC algorithm would perform very badly.

Laboratory

1. Based on the outline of the simple HDC algorithm, derive your version of the HDC algorithm in pseudocode which allows an easy implementation in Java (or C, C++).

2. Implement your version of HDC algorithm in Java. Use "MyHDC" as the name of your main class/program.

3. Provide two source files "good.source" and "bad.source", on which HDC would perform very well and very badly respectively. Indicate your definition of "good" and "bad" performance.

   [Hint] Define the input and the output of your (compression and decompression) algorithms first.

Sample examination questions

1. Describe with an example how a Run-Length Coder works.

2. Apply the HDC (Hardware Data Compression) algorithm to the sequence:

   `␣␣␣␣␣␣␣BC␣␣␣A␣1144330000␣␣EFGHHHH`

   Demonstrate the compressed output and explain the meaning of each control symbol.

Chapter 3

Preliminaries

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 2.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 1.

Huffman coding

Huffman coding is a successful compression method used originally for text compression. It assumes that each character is stored as a 8-bit ASCII code.

You may have already come across Huffman coding in a programming course. If so, review the following two questions:

1. What property of the text does Huffman coding algorithm require in order to fulfil the compression?
2. What is the main idea of Huffman coding?

Huffman coding works well on a text file for the following reasons:

- Characters are represented normally by fixed-length codewords[1] in computers. The codewords are often 8-bit long. Examples are ASCII code and EBCDIC code.

  Example 3.1 *In ASCII code, codeword* `p1000001` *represents character 'A';* `p1000010` *'B';* `p1000101` `'E'`*, etc., where* `p` *is the parity bit.*

- In any text, some characters occur far more frequently than others. For example, in English text, letters E,A,O,T are normally used much more frequently than J,Q,X.
- It is possible to construct a *uniquely decodable* code with variable codeword lengths.

[1] *Note: it is useful to distinguish the term 'codeword' from the term 'cord' although the two terms can be exchangeable sometimes. In this subject guide, a code consists of a number of codewords (see Example 3.1.)*

Our aim is to reduce the total number of bits in a sequence of `1`s and `0`s that represent the characters in a text. In other words, we want to reduce the average number of bits required for each symbol in the text.

Huffman's idea

Instead of using a fixed-length code for each symbol, Huffman's idea is to represent a frequently occurring character in a source with a shorter code and to represent a less frequently occurring one with a longer code. So for a text source of symbols with different frequencies, the total number of bits in this way of representation is, hopefully, significantly reduced. That is to say, the number of bits required for each symbol *on average* is reduced.

Example 3.2 *Frequency of occurrence:*

E	A	O	T	J	Q	X
5	5	5	3	3	2	1

Suppose we find a code that follows Huffman's approach. For example, the most frequently occurring symbol E *and* A *are assigned the shortest 2-bit codeword, and the lest frequently occurring symbol* X *is given a longer 4-bit codeword, and so on, as below:*

E	A	O	T	J	Q	X
10	11	000	010	011	0010	0011

Then the total number of bits required to encode string 'EEETTJX' is only $2+2+2+3+3+3+4 = 19$ *(bits). This is significantly fewer than* $8 \times 7 = 56$ *bits when using the normal 8-bit ASCII code.*

Huffman encoding algorithm

A frequency based coding scheme (algorithm) that follows Huffman's idea is called *Huffman coding*. Huffman coding is a simple algorithm that generates a set of variable-size codewords of the minimum average length. The algorithm for Huffman encoding involves the following steps:

1. Constructing a frequency table *sorted* in descending order.
2. Building a *binary tree*
   Carrying out iterations until completion of a complete binary tree:
   (a) Merge the last two items (which have the minimum frequencies) of the frequency table to form a new combined item with a sum frequency of the two.
   (b) Insert the combined item and update the frequency table.
3. Deriving *Huffman tree*
   Starting at the *root*, trace down to every *leaf*; mark '0' for a *left branch* and '1' for a *right* branch.
4. Generating Huffman code:
   Collecting the *0*s and *1*s for each path from the root to a leaf and assigning a 0-1 codeword for each symbol.

We use the following example to show how the algorithm works:

Example 3.3 *Compress 'BILL BEATS BEN.' (15 characters in total) using the Huffman approach.*

1. **Constructing the frequency table**

B	I	L	E	A	T	S	N	SP(space)	.	character
3	1	2	2	1	1	1	1	2	1	frequency

Sort the table in descending order:

```
B  L  E  SP I  A  T  S  N  .
3  2  2  2  1  1  1  1  1  1
```

2. **Building the binary tree**

There are two stages in each step:

- *(a)* *combine the last two items on the table*
- *(b)* *adjust the position of the combined item on the table so the table remains sorted.*

For our example, we do the following:

(a) *Combine*

```
B  L  E  SP I  A  T  S  (N.)
3  2  2  2  1  1  1  1  2
```

Update[2]

```
B  (N.)  L  E  SP I  A  T  S
3  2     2  2  2  1  1  1  1
```

[2] *Note:* (N.) *has the same frequency as* L *and* E*, but we have chosen to place it at the highest possible location - immediately after* B *(frequency 3).*

(b)
```
B  (TS) (N.)  L  E  SP I  A
3  2    2     2  2  2  1  1
```

(c)
```
B  (IA) (TS) (N.)  L  E  SP
3  2    2    2     2  2  2
```

(d)
```
(E SP) B  (IA) (TS) (N.)  L
4      3  2    2    2     2
```

(e)
```
((N.)  L)  (E SP) B  (IA) (TS)
4          4      3  2    2
```

(f)
```
((IA)(TS)) ((N.)  L)  (E SP) B
4          4          4      3
```

(g)
```
((E SP) B) ((IA)(TS)) ((N.)  L)
7          4          4
```

(h)
```
(((IA)(TS)) ((N.)  L)) ((E SP) B)
8                      7
```

(i)
```
((((IA)(TS)) ((N.)  L)) ((E SP) B))
15
```

The complete binary tree is:

```
   ((((IA)(TS)) ((N.)  L)) ((E SP) B))
                   /            \
   (((IA)(TS)) ((N.)  L))  ((E SP) B)
         /        \             /    \
((IA)(TS))      ((N.)  L)    (E SP)   B
    /   \         /    \       / \
 (IA)  (TS)     (N.)    L     E  SP
   / \  / \     / \
   I A  T S     N .
```

3. **Deriving Huffman tree**

```
       ((((IA)(TS)) ((N.)  L)) ((E SP) B))
                   0/            \1
       (((IA)(TS)) ((N.)  L))  ((E SP) B)
          0/        \1           0/    \1
   ((IA)(TS))      ((N.)  L)    (E SP)   B
     0/   \1       0/    \1      0/ \1
    (IA)  (TS)     (N.)    L      E   SP
    0/ \1 0/ \1   0/ \1
     I A   T S     N .
```

4. **Generating Huffman code**

```
     I    A    T    S    N    .    L   E   SP  B
     0000 0001 0010 0011 0100 0101 011 100 101 11

  x  1    1    1    1    1    1    2   2   2   3
     4    4    4    4    4    4    3   3   3   2
```

5. **Saving percentage**

Comparison of the use of Huffman coding and the use of 8-bit ASCII or EBCDIC Coding:

Huffman	*ASCII/EBCDIC*	*Saving bits*	*Percentage*
48	*120*	*72*	*60%*
		$120 - 48 = 72$	$72/120 = 60\%$

Decoding algorithm

The decoding process is based on the same Huffman tree. This involves the following types of operations:

- We read the coded message bit by bit. Starting from the root, we follow the bit value to traverse one edge down the the tree.
- If the current bit is 0 we move to the left child, otherwise, to the right child.
- We repeat this process until we reach a leaf. If we reach a leaf, we will decode one character and re-start the traversal from the root.
- Repeat this read-move procedure until the end of the message.

Example 3.4 *Given a Huffman-coded message,*
11100010010111110000001001000011101111000000110110101,
what is the decoded message?

```
       ((((IA)(TS)) ((N.)  L)) ((E SP) B))
                   0/            \1
       (((IA)(TS)) ((N.)  L))  ((E SP) B)
          0/        \1           0/    \1
   ((IA)(TS))      ((N.)  L)    (E SP)   B
     0/   \1       0/    \1      0/ \1
    (IA)  (TS)      N.     L      E   SP
    0/ \1 0/ \1   0/ \1
     I A   T S     N  .
```

After reading the first two 1s of the coded message, we reach the leaf B. Then the next 3 bits 100 lead us to the leaf E, and so on.

Finally, we get the decoded message: 'BEN BEATS BILL.'

Observation on Huffman coding

1. Huffman codes are not unique, for two reasons:
   (a) There are two ways to assign a 0 or 1 to an edge of the tree. In Example 3.3, we have chosen to assign 0 to the left edge and 1 for the right. However, it is possible to assign 0 to the right and 1 to the left. This would not make any difference to the compression ratio.
   (b) There are a number of different ways to insert a combined item into the frequency table. This leads to different binary trees. We have chosen in the same example to:
      i. make the item at the higher position the left child
      ii. insert the combined item on the frequency table at the highest possible position.
2. The Huffman tree built using our approach in the example tends to be more balanced than those built using other approaches. The code derived with our method in the example is called *canonical minimum-variance* Huffman code.

   The differences among the lengths of codewords in a canonical minimum-variance code turn out to be the minimum possible.
3. The frequency table can be replaced by a probability table. In fact, it can be replaced by any approximate statistical data at the cost of losing some compression ratio. For example, we can apply a probability table derived from a typical text file in English to any source data.
4. When the alphabet is small, a fixed length (less than 8 bits) code can also be used to save bits.

   [3] *i.e. the size is smaller or equal to 32.*

   Example 3.5 *If the size of the alphabet set is not bigger than 32*[3]*, we can use five bits to code each character. This would give a saving percentage of*

$$\frac{8 \times 32 - 5 \times 32}{8 \times 32} = 37.5\%.$$

Shannon-Fano coding

This is another approach very similar to Huffman coding. In fact, it is the first well-known coding method. It was proposed by C. Shannon (Bell Labs) and R. M. Fano (MIT) in 1940.

The Shannon-Fano coding algorithm also uses the probability of each symbol's occurrence to construct a code in which each codeword can be of different length. Codes for symbols with low probabilities are assigned more bits, and the codewords of various lengths can be uniquely decoded.

Shannon-Fano algorithm

Given a list of symbols, the algorithm involves the following steps:

1. Develop a frequency (or probability) table

2. Sort the table according to frequency (the most frequent one at the top)
3. Divide the table into 2 halves with similar frequency counts
4. Assign the upper half of the list a *0* and the lower half a *1*
5. Recursively apply the step of division (2.) and assignment (3.) to the two halves, subdividing groups and adding bits to the codewords until each symbol has become a corresponding leaf on the tree.

Example 3.6 *Suppose the sorted frequency table below is drawn from a source. Derive the Shannon-Fano code.*

```
Symbol        Frequency
-----------------------
  A             15
  B             7
  C             6
  D             6
  E             5
```

Solution

1. First division:

   (a) Divide the table into two halves so the sum of the frequencies of each half are as close as possible.

   ```
   Symbol        Frequency
   --------------------------
   A             15                            22
   B             7
   11111111111111111111111111 First division
   C             6                             17
   D             6
   E             5
   ```

   (b) Assign one bit of the symbol (e.g. upper group 0s and the lower 1s).

   ```
   Symbol        Frequency     Code
   --------------------------------
   A             15            0
   B             7             0
   11111111111111111111111111 First division
   C             6             1
   D             6             1
   E             5             1
   ```

2. Second division:
   Repeat the above recursively to each group.

```
Symbol        Frequency     Code
--------------------------------
A             15            00
22222222222222222222222222222 Second division
B             7             01
1111111111111111111111111 First division
C             6             10
22222222222222222222222222222 Second division
D             6             11
E             5             11
```

3. Third division:

```
Symbol        Frequency     Code
--------------------------------
A             15            00
22222222222222222222222222222 Second division
B             7             01
1111111111111111111111111 First division
C             6             10
22222222222222222222222222222 Second division
D             6             110
33333333333333333333333333333333 Third division
E             5             111
```

4. So we have the following code (consisting of 5 codewords) when the recursive process ends:

```
A   B   C   D    E
00  01  10  110  111
```

Observation on Shannon-Fano coding

1. It is not always easy to find the best division (see Step 3., the encoding algorithm).

2. The encoding process starts by assigning the most significant bits to each code and then works down the tree recursively until finished. It is as to construct a binary tree from the top. The following tree is for the example above:

```
      ABCDE
     0/   \1
    AB    CDE
  0/ \1  0/  \1
  A   B   C   DE
            0/  \1
            D    E
```

3. The decoding process is similar to Huffman decoding but the encoding process is not as simple and elegant as Huffman encoding.

Learning outcomes

On completion of your studies in this chapter, you should be able to:

- describe Huffman coding and Shannon-Fano coding
- explain why it is not always easy to implement Shannon-Fano algorithm
- demonstrate the encoding and decoding process of Huffman and Shannon-Fano coding with examples.

Activities

1. Derive a Huffman code for string AAABEDBBTGGG.
2. Derive a Shannon-Fano code for the same string.
3. Provide an example to show step by step how Huffman decoding algorithm works.
4. Provide a similar example for the Shannon-Fano decoding algorithm.

Laboratory

1. Derive a simple version of Huffman algorithm in pseudocode.
2. Implement your version of Huffman algorithm in Java (or C, C++).
3. Similar to Lab2, provide two source files: the good and the bad. Explain what you mean by good or bad.
4. Implement the Shannon-Fano algorithm.
5. Comment on the difference the Shannon-Fano and the Huffman algorithm.

Sample examination questions

1. Derive step by step a canonical minimum-variance Huffman code for alphabet {A, B, C, D, E, F}, given that the probabilities that each character occurs in all messages are as follows:

```
Symbol    Probability
-------   -----------
  A          0.3
  B          0.2
  C          0.2
  D          0.1
  E          0.1
  F          0.1
```

2. Compute the average length of the Huffman code derived from the above question.
3. Given $\mathcal{S} = \{A, B, C, D, E, F, G, H\}$ and the symbols' occurring probabilities 0.25, 0.2, 0.2, 0.18, 0.09, 0.05, 0.02, 0.01, construct a canonical minimum-variance Huffman code for this input.

Chapter 4

Coding symbolic data

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 2.3-2.5.

Further reading

Sayood, Khalid *Introduction to Data Compression* (Morgan Kaufmann, 1996) [ISBN 1-55860-346-8]. Chapter 2.

In this chapter, we shall look more closely at the structure of compression algorithms in general. Starting with symbolic data compression, we apply the information theory to gain a better understanding of compression algorithms. Some conclusions we draw from this chapter may also be useful for multimedia data compression in later chapters.

Compression algorithms

You will recall that in the Introduction, we said that data compression essentially consists of two types of work: modelling and coding. It is often useful to consciously consider the two entities of compression algorithms separately.

- The general model is the embodiment of what the compression algorithm knows about the source domain. Every compression algorithm has to make use of some knowledge about its platform.

  Example 4.1 *Consider the Huffman encoding algorithm. The model is based on the probability distribution of characters of a source text.*

- The device that is used to fulfil the task of coding is usually called *coder* meaning *encoder*. Based on the model and some calculations, the coder is used to
  - derive a code
  - encode (compress) the input.

  Example 4.2 *Consider the Huffman encoding algorithm again. The coder assigns shorter codes to the more frequent symbols and longer codes to infrequent ones.*

A similar structure applies to decoding algorithms. There is again a *model* and a *decoder* for any decoding algorithm.

Conceptually, we can distinguish *two* types of compression algorithms, namely, *static*, or *adaptive* compression, based on whether the model structure may be updated during the process of compression or decompression.

The model-coder structures can be seen clearly in diagrams Figure 4.1 and Figure 4.2:

- **Static (non-adaptive) system** (Figure 4.1): This model remains unchanged during the compression or decompression process.

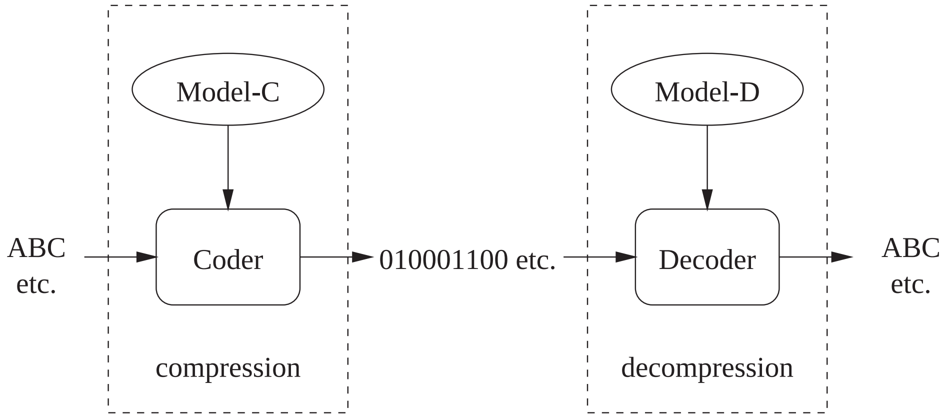

Figure 4.1: A static or non-adaptive compression system

- **Adaptive system** (Figure 4.2): The model may be changed during the compression or decompression process according to the change of input (or feedback from the output).
  Some adaptive algorithms actually build the model based on the input starting from an empty model.

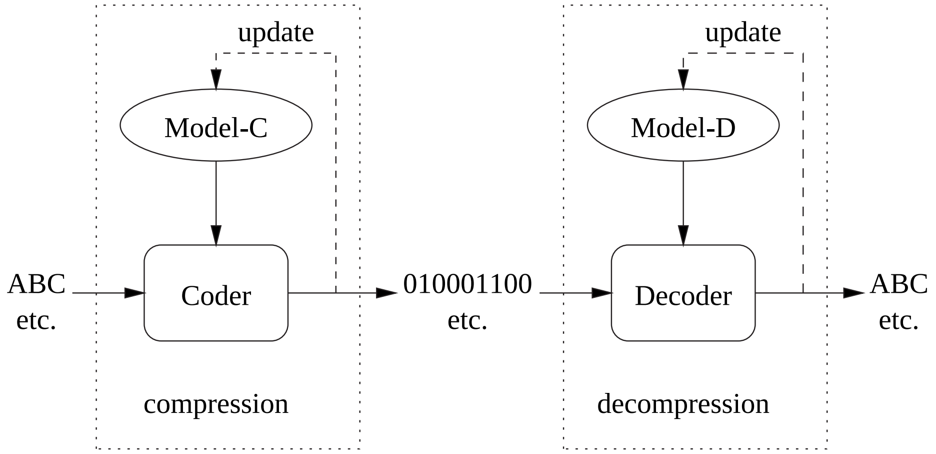

Figure 4.2: Adaptive compression system

In practice, the software or hardware for implementing the model is often a mixture of static and adaptive algorithms, for various reasons such as efficiency.

Symmetric and asymmetric compression

In some compression systems, the model for compression (Model-C in the figures) and that for decompression (Model-D) are identical. If they are identical, the compression system is called *symmetric*, otherwise, it is said to be *non-symmetric*. The compression using a symmetric system is called *symmetric compression*, and the compression using an asymmetric system is called *asymmetric compression*.

Coding methods

In terms of the length of codewords used *before* or *after* compression, compression algorithms can be classified into the following categories:

1. **Fixed-to-fixed**: each symbol before compression is represented by a fixed number of bits (e.g. 8 bits in ASCII format) and is encoded as a sequence of bits of a fixed length after compression.

   Example 4.3 *A:00, B:01, C:10, D:11* [1]

2. **Fixed-to-variable**: each symbol before compression is represented by a fixed number of bits and is encoded as a sequence of bits of different length.

   Example 4.4 *A:0; B:10; C:101; D:0101.*

3. **Variable-to-fixed**: a sequence of symbols represented in different number of bits before compression is encoded as a fixed-length sequence of bits.

   Example 4.5 *ABCD:00; ABCDE:01; BC:11.*

4. **Variable-to-variable**: a sequence of symbols represented in different number of bits before compression is encoded as a variable-length sequence of bits.

   Example 4.6 *ABCD:0; ABCDE:01; BC:1; BBB:0001.*

[1] *For ease of read, the symbol itself is used here instead of its ASCII codeword.*

Question 4.1 *Which class does Huffman coding belong to?*

Solution It belongs to the *fixed-to-variable* class. Why? Each symbol before compression is represented by a fixed length code, e.g. 8 bits in ASCII, and the codeword for each symbol after compression consists of different number of bits.

Question of unique decodability

The issue of unique decodability arises during the decompression process when a variable length code is used. Ideally, there is only one way to decode a sequence of bits consisting of codewords. However, when symbols are encoded by a variable-length code, there may be more than one way to identifying the codewords from the sequence of bits.

Given a variable length code and a sequence of bits to decompress, the code is regarded as *uniquely decodable* if there is only one possible way to decode the bit sequence in terms of the codewords.

Example 4.7 *Given symbols A, B, C and D, we wish to encode them as follows: A:0; B:10; C:101; D:0101. Is this code uniquely decodable?*

The answer is 'No'. Because an input such as '0101101010' can be decoded in more than one way, for example, as ACCAB or as DCAB.

However, for the example above, there is a solution if we introduce a new symbol to separate each codeword. For example, if we use a 'stop' symbol "/". We could then encode DDCAB as '0101/0101/101/0/10'. At the decoding end, the sequence '0101/0101/101/0/10' will be easily decoded uniquely.

Unfortunately, the method above is too costly because of the extra symbol "/" introduced. Is there any alternative approach which allows us to uniquely decode a compressed message using a code with various length codewords? After all, how would we know whether a code with various length codewords is uniquely decodable?

Well, one simple solution is to find another code which is a so-called *prefix code* such as (0,11,101,1001) for (A,B,C,D).

Prefix and dangling suffix

Let us look at some concepts first:

- Prefix: Consider two binary codewords w_1 and w_2 with lengths k and n bits respectively, where $k < n$. If the first k bits of w_2 are identical to w_1, then w_1 is called a *prefix* of w_2.
- Dangling Suffix: The remain of last $n - k$ bits of w_2 is called the *dangling suffix*.

Example 4.8 *Suppose* $w_1 = 010$, $w_2 = \underline{010}11$. *Then the prefix of* w_2 *is 010 and the suffix is 11.*

Prefix codes

A prefix code is a code in which *no* codeword is a prefix to another codeword (Note: meaning 'prefix-free codes').
This occurs when no codeword for one symbol is a prefix of the codeword for another symbol.

Example 4.9 *The code (1, 01, 001, 0000) is a prefix code since no codeword is a prefix of another codeword.*

The code (0, 10, 110, 1011) is not a prefix code since 10 is a prefix of 1011.

Prefix codes are important in terms of uniquely decodability for the following two main reasons (See Sayood(2000), section 2.4.3 for the proof).

1. Prefix codes are uniquely decodable.

   This can be seen from the following informal reasoning:

   Example 4.10 *Draw a 0-1 tree for each code above, and you will see the difference. For a prefix code, the codewords are only associated with the leaves.*

```
      root                         root
    0/    \1                     0/    \1
   0/ \1   A                    A     0/  \1
 0/ \1  B                            (B)  0/
0/   C                                \1    C
D                                      \1
                                        D
```

2. For any non-prefix code whose codeword lengths satisfy certain condition (see section 'Kraft-McMillan inequality' below), we can always find a prefix code with the same codeword length distribution.

   Example 4.11 *Consider code (0, 10, 110, 1011).*

Kraft-McMillan inequality

Theorem 4.1 *Let C be a code with N codewords with lengths $l_1, l_2, \cdots, l_N$. If C is uniquely decodable, then*

$$K(C) = \sum_{i=1}^{N} 2^{-l_i} \leq 1$$

This inequality is known as the Kraft-McMillan inequality. (See Sayood(2000), section 2.4.3 for the proof).

In $\sum_{i=1}^{N} 2^{-l_i} \leq 1$, N is the number of codewords in a code, l_i is the length of the ith codeword.

Example 4.12 *Given an alphabet of 4 symbols (A, B, C, D), would it be possible to find a uniquely decodable code in which a codeword of length 2 is assigned to A, length 1 to B and C, and length 3 to D?*

Solution Here we have $l_1 = 2$, $l_2 = l_3 = 1$, and $l_4 = 3$.

$$\sum_{i=1}^{4} 2^{-l_i} = \frac{1}{2^2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2^3} > 1$$

Therefore, we cannot hope to find a uniquely decodable code in which the codewords are of these lengths.

Example 4.13 *If a code is a prefix code, what can we conclude about the lengths of the codewords?*

Solution Since prefix codes are uniquely decodable, they must satisfy the Kraft-McMillan Inequality.

Some information theory

The information theory is based on mathematical concepts of probability theory. The term *information* carries a sense of unpredictability in transmitted messages. The *information source* can be represented by a set of event symbols (random variables) from which the information of each event can be measured by the surprise that the event may cause, and by the probability rules that govern the emission of these symbols.

The symbol set is frequently called the *source alphabet*, or *alphabet* for short. The number of elements in the set is called *cardinality* ($|\mathcal{A}|$).

Self-information

This is defined by the following mathematical formula:

$$I(A) = -\log_b P(A),$$

where A is an event, $P(A)$ is the probability that event A occurs.

The logarithm base (i.e. b in the formula) may be in:

- unit *bits*: base 2 (used in the subject guide)
- unit *nats*: base e
- unit *hartleys*: base 10.

The self-information of an event measures the amount of one's surprise evoked by the event. The negative logarithm $-\log_b P(A)$ can be written as

$$\log_b \frac{1}{P(A)}.$$

Note that $\log(1) = 0$, and that $|-\log(P(A))|$ increases as $P(A)$ decreases from 1 to 0. This supports our intuition from daily experience. For example, a low-probability event tends to cause more surprise.

Entropy

The precise mathematical definition of this measure was given by Shannon. It is called *entropy of the source* which is associated with the experiments on the (random) event set.

$$H = \sum P(A_i)I(A_i) = -\sum P(A_i)\log_b P(A_i),$$

where source $|\mathcal{A}| = (A_1, \ldots, A_N)$.

The information content of a source is an important attribute. Entropy describes the average amount of information converged per source symbol. This can also be thought to measure the expected amount of surprise caused by the event.

If the experiment is to take out the symbols A_i from a source $\mathcal{A}$, then

- the entropy is a measure of the minimum average number of binary symbols (bits) needed to encode the output of the source.
- Shannon showed that the best that a lossless symbolic compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

Example 4.14 *Consider the three questions below:*

1. *Given four symbols A, B, C and D, the symbols occur with an equal probability. What is the entropy of the distribution?*
2. *Suppose they occur with probabilities 0.5, 0.25, 0.125 and 0.125 respectively. What is the entropy associated with the event (experiment)?*
3. *Suppose the probabilities are 1,0,0,0. What is the entropy?*

Solution

1. The entropy is $1/4(-\log_2(1/4)) \times 4 = 2$ bits

2. The entropy is $0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3 = 1.75$ bits
3. The entropy is 0 bit.

Optimum codes

In information theory, the ratio of the entropy of a source to the average number of binary bits used to represent the source data is a measurement of the information *efficiency* of the source. In data compression, the ratio of the entropy of a source to the average length of the codewords of a code can be used to measure how successful the code is for compression.

Here a source is usually described as an alphabet $\alpha = \{s_1, \cdots, s_n\}$ and the next symbol chosen randomly from α is s_i with probability $Pr[s_i] = p_i$, $\sum_{i=1}^{n} p_i = 1$.

Information Theory says that *the best that a lossless symbolic compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.*

We write done the entropy of the source:

$$\sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} = p_1 \log_2 \frac{1}{p_1} + p_2 \log_2 \frac{1}{p_2} + \cdots + p_n \log_2 \frac{1}{p_n}$$

Using a variable length code to the symbols, l_i bits for s_i , the average number of bits is

$$\bar{l} = \sum_{i=1}^{n} l_i p_i = l_1 p_1 + l_2 p_2 + \cdots + l_n p_n$$

The code is optimum if the average length equals to the entropy. Let

$$\sum_{i=1}^{n} l_i p_i = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

and rewrite it as

$$\sum_{i=1}^{n} p_i l_i = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

This means that it is *optimal* to encode each s_i with $l_i = -\log_2 p_i$ bits, where $1 \leq i \leq n$.

It can be proved that:

1. the average length of any uniquely decodable code, e.g. prefix codes must be $\geq$ the entropy
2. the average length of a uniquely decodable code is equal to the entropy *only* when, for all i, $l_i = -log_2 p_i$, where l_i is the length and p_i is the probability of codeword$_i$[2].
3. the average codeword length of the Huffman code for a source is greater and equal to the entropy of the source and less than the entropy plus 1. [Sayood(2000), section 3.2.3]

[2] *This can only happen if all probabilities are negative powers of 2 in Huffman codes, for l_i has to be an integer (in bits).*

Scenario

What do the concepts such as *variable length codes*, *average length of codewords* and *entropy* mean in practice?

Let us see the following scenario: Ann and Bob are extremely good friends but they live far away to each other. They are both very poor students financially. Ann wants to send a shortest message to Bob in order to save her money. In fact, she decides to send to Bob a *single* symbol from their own secret alphabet. Suppose that:

- the next symbol that Ann wants to send is randomly chosen with a known probability
- Bob knows the alphabet and the probabilities associated with the symbols.

Ann knows that you have studied the Data compression so she asks you the important questions in the following example:

Example 4.15 *Consider the three questions as below:*

1. *To minimise the average number of bits Ann uses to communicate her symbol to Bob, should she assign a fixed length code or a variable length code to the symbols?*
2. *What is the average number of bits needed for Ann to communicate her symbol to Bob?*
3. *What is meant by a 0 entropy? For example, what is meant if the probabilities associated with the alphabet are* $\{0,0,1, \cdots, 0\}$*?*

You give Ann the following:

Solution

1. She should use variable length codes because she is likely to use some symbols more frequently than others. Using variable length codes can save bits hopefully.
2. Ann needs at least the average number of bits that equal to the entropy of the source. That is -$\sum_{i=1}^{n} p_i \log_2 p_i$ bits.
3. A '0 entropy' means that the minimum average number of bits that Ann needs to send to Bob is zero.

   Probability distribution $\{0,0,1, \cdots, 0\}$ means that Ann will definitely send the third symbol in the alphabet as the next symbol to Bob and Bob knows this.

   If Bob knows what Ann is going to say then she does not need to say anything, does she?!

Learning outcomes

On completion of your studies in this chapter, you should be able to:

- explain why modelling and coding are usually considered separately for compression algorithm design
- identify the model and the coder in a compression algorithm
- distinguish a static compression system by an adaptive one
- identify prefix codes
- demonstrate the relationship between prefix codes, Kraft-McMillan inequality and the uniquely decodability
- explain how entropy can be used to measure the code optimum.

Activities

1. Given an alphabet {a, b} with Pr[a]= 1/5 and Pr[b]= 4/5. Derive a canonical minimum variance Huffman code and compute:
   (a) the expected average length of the Huffman code
   (b) the entropy distribution of the Huffman code.
2. What is a prefix code? What can we conclude about the lengths of a prefix code? Provide an example to support your argument.
3. If a code is *not* a prefix code, can we conclude that it will not be uniquely decodable? Give reasons.
4. Determine whether the following codes are uniquely decodable:
   (a) {0,01,11,111}
   (b) {0,01,110,111}
   (c) {0,10,110,111}
   (d) {1,10,110,111}.

Laboratory

1. Design and implement a method `entropy` in Java which takes a set of probability distribution as the argument and returns the entropy of the source.
2. Design and implement a method `averageLength` in Java which takes two arguments: (1) a set of length of a code; (2) the set of the probability distribution of the codewords. It then returns the average length of the code.

Sample examination questions

1. Describe briefly how each of the two classes of lossless compression algorithms, namely the *adaptive* and the *non-adaptive*, works in its model. Illustrate each with an appropriate example.
2. Determine whether the following codes for {A, B, C, D} are *uniquely decodable*. Give your reasons for each case.
   (a) {0, 10, 101, 0101}
   (b) {000, 001, 010, 011}
   (c) {00, 010, 011, 1}
   (d) {0, 001, 10, 010}
3. Lossless coding models may further be classified into *three* types, namely *fixed-to-variable*, *variable-to-fixed*, *variable-to-variable*.

   Describe briefly the way that each encoding algorithm works in its model. Identify one example of a well known algorithm for each model.
4. Derive step by step a canonical minimum-variance Huffman code for alphabet {A, B, C, D, E, F}, given that the probabilities that each character occurs in all messages are as follows:

```
Symbol    Probability
-------   -----------
A         0.3
B         0.2
C         0.2
D         0.1
E         0.1
F         0.1
```

Compare the *average length* of the Huffman code to the *optimal length* derived from the entropy distribution. Specify the unit of the codeword lengths used.

Hint: $\log_{10} 2 \approx 0.3$; $\log_{10} 0.3 \approx -0.52$; $\log_{10} 0.2 \approx -0.7$; $\log_{10} 0.1 = -1$;

5. Determine whether the code {0, 10, 011, 110, 1111} is a prefix code and explain why.

6. If a code is a prefix code, what can we conclude about the lengths of the codewords?

7. Consider the alphabet {A, B}. Suppose the probability of A and B, $\Pr[A]$ and $\Pr[B]$ are 0.2 and 0.8 respectively. It has been claimed that even the best canonical minimum-variance Huffman coding is about 37% worse than its optimal binary code. Do you agree with this claim? If Yes, demonstrate how this result can be derived step by step. If No, show your result with good reasons.

   Hint: $\log_{10} 2 \approx 0.3$; $\log_{10} 0.8 \approx -0.1$; $\log_{10} 0.2 \approx -0.7$.

8. Following the above question,

   (a) derive the alphabet that is expanded by grouping 2 symbols at a time;

   (b) derive the canonical Huffman code for this expanded alphabet.

   (c) compute the expected average length of the Huffman code.