UNIVERSITY
OF LONDON

Data compression

I. Pu

CO3325

2004

Undergraduate study in
Computing and related programmes

Goldsmiths
UNIVERSITY OF LONDON

This guide was prepared for the University of London by:

I. Pu

This is one of a series of subject guides published by the University. We regret that due to pressure of work the author is 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.

In this and other publications you may see references to the 'University of London International Programmes', which was the name of the University's flexible and distance learning arm until 2018. It is now known simply as the 'University of London', which better reflects the academic award our students are working towards. The change in name will be incorporated into our materials as they are revised.

University of London
Publications Office
32 Russell Square
London WC1B 5DN
United Kingdom

london.ac.uk

Published by: University of London

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
   ```

Notes

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`G`$r_6 n_6$`BCDEFG`$r_2 n_9$`55␣LM`r_{12}`7`

Solution

1. The first 3 `G`s are read and encoded by r_3`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`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`55GHJK␣LM`.
6. The next 12 '7's are found and encoded by r_{12}`7`.

Therefore the encoded output is: r_3`G`$r_6 n_6$`BCDEFG`$r_2 n_9$`55␣LM`r_{12}`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
```

(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 .
```

[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).*

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, 11100010010111100000100100011101110000011011010 1, 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	
1111111111111111111111111 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
1111111111111111111111111 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
22222222222222222222222222 Second division
B             7             01
11111111111111111111111 First division
C             6             10
222222222222222222222222222 Second division
D             6             11
E             5             11
```

3. Third division:

```
Symbol        Frequency     Code
--------------------------------
A             15            00
22222222222222222222222222 Second division
B             7             01
11111111111111111111111 First division
C             6             10
222222222222222222222222222 Second division
D             6             110
33333333333333333333333333333 Third division
E             5             111
```

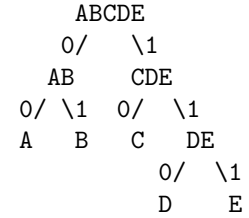
4. So we have the following code (consisting of 5 codewords) when the recursive process ends:

```
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:



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.

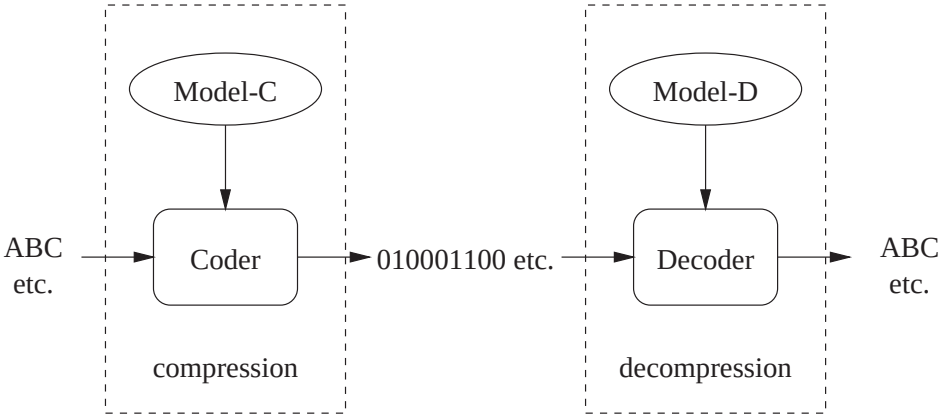


Figure 4.1: A static or non-adaptive compression system

- **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.

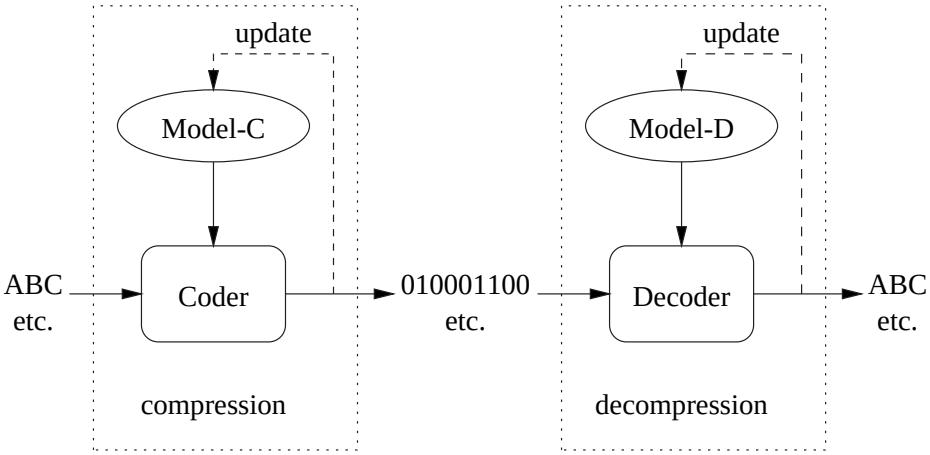


Figure 4.2: Adaptive compression system

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.*

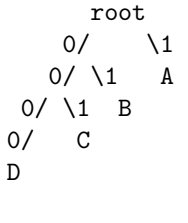
The code (0, 10, 110, 1011) is not a prefix code since 10 is a prefix of 1011.

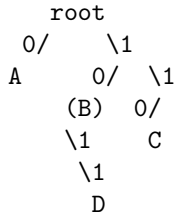
Prefix codes are important in terms of uniquely decodability for the following two main reasons (See Sayood(2000), section 2.4.3 for the proof).

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.*





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.

Chapter 5

Huffman coding

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.

In this chapter, we formally study the Huffman coding algorithms and apply the theory in Chapter 4.

Static Huffman coding

The *static* Huffman coding[1] is based on the following model:

- It includes a sequence of symbols from an alphabet $\alpha = \{s_1, s_2, \cdots, s_n\}$, where symbol s_i appears with a fixed probability p_i.
- The symbols are generated independently (which is called 'memoryless'). This means that the probability of finding s_i in the next character position is always p_i, regardless of what went before.

[1] *Assume that the fixed probability distribution is known to both the compressor and the decompressor.*

Huffman algorithm

You may recall from the previous studies that the Huffman compression algorithm contains the following main operations:

- Build a binary tree where the leaves of the tree are the symbols to be coded.
- The edges of the tree are labelled by a 0 or 1.

The decompression algorithm involves the operations where the codeword for a symbol is obtained by 'walking' down from the root of the tree to the leaf for each symbol.

Example 5.1 $\alpha = \{A, B, C, D, E, F\}$ *with probabilities 0.25, 0.2, 0.15, 0.15, 0.15, 0.1.*

Building the binary tree

1. Repeat the following until there is only one symbol in the alphabet:
   (a) If there is one symbol, the tree is the root and the leaf. Otherwise, take two symbols in alphabet s and s' which have the lowest probabilities p and p'.
   (b) Remove s and s' from the alphabet and add a new symbol $[s, s']$ with probability $p + p'$. Now the alphabet has one fewer symbol than before.
2. Recursively construct the tree, starting from the root.

Canonical and minimum-variance

As we know:

- there can be items with equal probabilities
- the roles of 0 and 1 are interchangeable.

To avoid this, we set the following rules:

1. A newly-created element is placed at the highest possible position in the list while keeping the list sorted
2. When combining two items, the one higher up in the list is assigned 0 and the lower down 1.

The code that follows these rules is called *canonical*.

By *minimum-variance*, we meant that the variation in the lengths of the codes is minimised.

Huffman coding that follows these rules is called *Canonical and Minimum-variance Huffman Coding*.

Implementation efficiency

The algorithm described earlier requires the system to:

- maintain the list of current symbols in decreasing order of probability,
- search the right place to insert the new symbols, which gives an $O(n^2)$ worst-case, where n is the number of symbols.

One way to solve the efficiency problem is to do the following:

- Maintain 2 probability (or frequency) lists: one (L_i) contains the original symbols in decreasing order of probability. The other (L_c) only contains 'super-symbols' which are obtained by combining symbols, initially empty.
- A new combined symbol will always go to the *front* of list L_c (in $O(1)$ time).
- Each time look at the ends of L_i and L_c to determine which symbols next to combine. The next combination may be two super-symbols or two initial symbols or a super-symbol with an initial symbol.

Example 5.2 *Show how the algorithm works when the alphabet is {A,B,C,D,E,F,G,H,I,J} and the probabilities are (in %) 19, 17, 15, 13, 11, 9, 7, 5, 3, 1.*

1.
```
Li: A   B   C   D   E   F   G   H   I   J
    19  17  15  13  11  9   7   5   3   1
Lc: Empty
```

2.
```
Li: A   B   C   D   E   F   G   H
    19  17  15  13  11  9   7   5
Lc: (IJ)
    4
```

3.
```
Li: A   B   C   D   E   F   G
    19  17  15  13  11  9   7
Lc: (H (IJ))
    9
```

4.
```
Li: A   B   C   D   E
    19  17  15  13  11
Lc: (FG) (H (IJ)))
    16    9
```

5.
```
Li: A   B   C   D
    19  17  15  13
Lc: (E (H (IJ))) (FG)
    20           16
```

6.
```
Li: A   B
    19  17
Lc: (CD) (E (H (IJ))) (FG)
    28   20           16
```

7.
```
Li: A
    19
Lc: (B (FG)) (CD) (E (H (IJ)))
    33       28   20
```

8.
```
Li: Empty
Lc: (A (E (H (IJ)))) (B (FG)) (CD)
    39               33       28
Li: Empty
```

9.
```
Lc: ((B (FG)) (CD)) (A (E (H (IJ))))
    61              39
Li: Empty
```

10.
```
Lc: (((B (FG)) (CD)) (A (E (H (IJ)))))
    100
Li: Empty
```

Construct the binary tree recursively from the root:

1.
```
      (((B (FG)) (CD)) (A (E (H (IJ)))))
        0 /                          \ 1
((B (FG)) (CD))                 (A (E (H (IJ))))
```

2.
```
     (((B (FG)) (CD)) (A (E (H (IJ)))))
         0 /                    \ 1
    0 /        \ 1          0 /     \ 1
(B (FG))       (CD)          A    (E (H (IJ)))
```

3.
```
      (((B (FG)) (CD)) (A (E (H (IJ)))))
           0 /                    \ 1
      0 /        \ 1          0 /     \ 1
  0 /     \ 1    0 / \ 1        A   0 /  \ 1
   B     (FG)     C   D              E  (H (IJ))
```

4.
```
      (((B (FG)) (CD)) (A (E (H (IJ)))))
           0 /                    \ 1
      0 /        \ 1          0 /     \ 1
  0 /     \ 1    0 / \ 1        A   0 /  \ 1
   B   0 / \ 1    C   D              E  0/ \1
        F   G                         H  0/ \1
```

So the code is:

```
B 000  F 0010  G 0011  C 010    D 011    A 10
E 110  H 1110  H 1110  I 11110  J 11111
```

Observation

Note that:

- Huffman coding belongs to the *fixed-to-variable* coding method.
- Huffman codes are *prefix codes* (meaning 'prefix-free codes'), so they are uniquely decodable.
- Huffman codes are optimal when probabilities of the symbols are all negative powers of 2, for l_i has to be an integer (in bits).
- Huffman codes are fragile for decoding: the entire file could be corrupted even if there is a 1-bit error.

A problem in Huffman codes

Remember that Huffman codes meet the entropy bound only when all probabilities are powers of 2. What if the alphabet is binary ($\alpha = \{a, b\}$)? The only optimal case[2] is when $p_a = 1/2$ and $p_b = 1/2$. Hence, Huffman codes can be bad.

[2] *Here we mean the average number of bits of a code equals to the entropy.*

Example 5.3 *Consider a situation when $p_a = 0.8$ and $p_b = 0.2$.*

Since Huffman coding needs to use 1 bit per symbol at least, to code the input, the Huffman codewords are 1 bit per symbol on average:

$$1 \times 0.8 + 1 \times 0.2 = 1 \text{ bits.}$$

However, the entropy of the distribution is

$$-(0.8 \log_2 0.8 + 0.2 \log_2 0.2) = 0.72 \text{ bits.}$$

This gives a gap of $1 - 0.72 = 0.28$ bits. The performance of Huffman encoding algorithm is, therefore, $0.28/1 \approx 28\%$ worse than optimal in this case.

Extended Huffman coding

The idea of extended Huffman coding is to *artificially* increase the alphabet size. For example, instead of assigning a codeword to every symbol, we derive a codeword for every 2 symbols.

The following example shows how to achieve this:

Example 5.4 *Create a new alphabet $\Sigma = \{aa, ab, ba, bb\}$ extended from $\{a,b\}$. Let aa be A, ab for B, ba for C and bb for D. We now have an extended alphabet $\Sigma = \{A, B, C, D\}$.*

The probability distribution can be calculated[3]*:*

[3] *Suppose that a or b occurs independently.*

$$Pr[A] = Pr[a] \times Pr[a] = 0.64$$
$$Pr[B] = Pr[a] \times Pr[b] = 0.16$$

$$Pr[C] = Pr[b] \times Pr[a] = 0.16$$
$$Pr[D] = Pr[b] \times Pr[b] = 0.04$$

We then follow the normal static Huffman encoding algorithm to derive the Huffman code for Σ (See previous chapters for examples).

The canonical minimum-variance code for this is `A=0, B=11, C=100, D=101`.

The average length is 1.56 bits for two symbols.

The original output became $1.56/2 = 0.78$ bits per symbol. This is only $(0.78 - 0.72)/0.78 \approx 8\%$ worse than optimal.

Learning outcomes

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

- explain how to improve the implementation efficiency of static Huffman encoding algorithm by maintaining two sorted probability lists
- illustrate some obvious weaknesses of Huffman coding
- describe how to narrow the gap between the *minimum average number of binary bits* of a code and the *entropy* using extended Huffman encoding method.

Activities

1. Given an alphabet $\mathcal{S} = \{A, B, C, D, E, F, G, H\}$ of symbols with the probabilities 0.25, 0.2, 0.2, 0.18, 0.09, 0.05, 0.02, 0.01 respectively in the input, construct a canonical minimum-variance Huffman code for the symbols.

2. Construct a canonical minimum-variance code for the alphabet A, B, C, D with probabilities 0.4, 0.3, 0.2 and 0.1. If the coded output is `101000001011`, what was the input?

3. Given an alphabet {a, b} with $\Pr[a] = 0.8$ and $\Pr[b] = 0.2$, determine the average code length if we group 3 symbols at a time.

4. Explain with an example how to improve the entropy of a code by grouping the alphabet.

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. Explain how the implementation efficiency of a canonical minimum-variance Huffman coding algorithm can be improved by means of maintaining two frequency lists.

2. Derive step by step a canonical minimum-variance Huffman code for alphabet {A, B, C, D, E, F} using the *efficient* implementation approach, 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

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

4. For 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.

Chapter 6

Adaptive Huffman coding

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 5.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 1.5.

Adaptive Huffman approach

The main characteristics of this approach include the following:

- The model calculates the probabilities of symbols as it goes along.
- For discussion convenience, we use the term *weights* instead of probabilities. The *weight* of a symbol is defined as the number of times that symbol has been seen (i.e. the *frequency* of its occurrence) so far.
- The output of the encoding (or decoding) process is generated immediately each time after reading in a symbol.
- The codes include Huffman codes as well as some fix-length codes. In the encoding process, for example, the model outputs a fix-length codeword (e.g. ASCII code) if the input symbol has been seen for the *first* time. Otherwise, it outputs a Huffman codeword.

Let the alphabet be $\alpha = \{\dagger, \sigma_1, \sigma_2, \cdots, \sigma_n\}$, and $g(\sigma_i)$ be any fixed-length codeword for σ_i (e.g. ASCII code). In order to indicate whether the next output codeword is a fix-length or a variable-length codeword, one special symbol $\dagger$ ($\notin \alpha$) is defined as a *shift* key solely for communication between the compressor and decompressor.

Compressor

The compression algorithm maintains a subset of symbols S of some alphabet (i.e. $S \in \alpha$)) that the model has seen so far. A Huffman code (i.e. the tree) for all the symbols in S is also maintained. The weight of $\dagger$ is always 0. The weight of any other symbol in S is its frequency so far.

Initially, $S = \{\dagger\}$ and the Huffman tree has the single node of symbol $\dagger$. During the process, the Huffman tree is used to assign codes to the symbols in S and updated after each output.

Let $h(\sigma_i)$ be the current Huffman code for σ_i and `DAG` for the special symbol $\dagger$ in the algorithm below.

Encoding algorithm

The following encoding algorithm shows how the compressor works:

```
1. Initialise the Huffman tree T containing
      the only node DAG.
2. while (more characters remain) do
      s:= next_symbol_in_text();
      if (s has been seen before) then output h(s)
      else output h(DAG) followed by g(s)
      T:= update_tree(T);
   end
```

`next_symbol_in_text()` is a function that reads one symbol from the input sequence. `update_tree` is another function which does the following:

Function `update_tree`

```
1. If s is not in S, then add s to S; weight_s:=1;
   else weight_s:=weight_s + 1;
2. Recompute the Huffman tree for the new set
      of weights and/or symbols.
```

Decompressor

The decompression algorithm maintains a subset of real symbols S' that the system has seen so far. The weight of † is always 0, and the weight of any other symbol is the frequency of occurrence so far in the decoded output. Initially, $S' := \{\dagger\}$.

Decoding algorithm

```
1. Initialise the Huffman tree T
      with single node DAG
2. while (more bits remain) do
      s:=huffman_next_sym();
      if (s==DAG) then
         s:= read_unencoded_sym();
      else output s;
      T:=update_tree(T);
   end
```

Function `huffman_next_sym()`

The function reads bits from the input until it reaches a leaf node and returns the symbol with which that leaf is labelled.

```
1. start at root of Huffman tree;
2. while (not reach leaf) do
      read_next_bit();
      follow edge;
   end
3. return the symbol of leaf reached.
```

Function `read_unencoded_sym()`

The function simply reads the next *unencoded* symbol from the input. For example, if the original encoding was an ASCII code, then it would read the next 7 bits (or 8 bits including the parity bit).

As you saw earlier (in section 'Compressor'), the function `update_tree` does the following:

```
1. If s is not in S, then add s to S
   else add 1 to the weight of s;
2. Recomputes the Huffman tree for
   the new set of weights and/or symbols.
```

Observation

Huffman coding, either static or adaptive, has two disadvantages that remain unsolved:

Problem 1 It is not optimal unless *all* probabilities are negative powers of 2. This means that there is a gap between the average number of bits and the entropy in most cases.

Recall the particularly bad situation for binary alphabets. Although by grouping symbols and extending the alphabet, one may come closer to the optimal, the blocking method requires a larger alphabet to be handled. Sometimes, extended Huffman coding is not that effective at all.

Problem 2 Despite the availability of some clever methods for counting the frequency of each symbol reasonably quickly, it can be very *slow* when rebuilding the entire tree for each symbol. This is normally the case when the probability distributions change rapidly with each symbol.

Learning outcomes

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

- describe how adaptive Huffman coding algorithms work with examples
- identify implementation issues of Huffman coding algorithms
- explain the two problems (weaknesses) of the Huffman coding algorithms in general.

Activities

1. Explain, with an example, how adaptive Huffman coding works.
2. Trace the adaptive Huffman coding algorithms to show how the following sequence of symbols is encoded and decoded:

   `aaabbcddbbccc`

Laboratory

1. Design and implement a simple version of Adaptive Huffman encoding algorithm.
2. Design and implement a simple version of Adaptive Huffman decoding algorithms.

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. Show how to encode the sequence below step by step using the adaptive Huffman coding algorithm.

   `abcbbdaaddd`

Chapter 7

Arithmetic coding

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 4.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 1.7.

Arithmetic coding

Arithmetic coding is the only replacement whose benefits have been successfully demonstrated after 15 years of Huffman coding. It bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. However, more bits are needed in the output number for longer, complex messages.

The output or Arithmetic coding is essentially a sequence of *decimal* digits, a fraction in the interval [0,1)[1]. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction.

[1] *[0,1) means all real numbers* ≥ 0 *and* < 1.

The arithmetic coding algorithm overcomes the problems in Huffman coding discussed at the end of Chapter 6:

- (for Problem 1) encodes *a sequence* of symbols at a time, instead of a single symbol.
- (for Problem 2) it outputs the coded version only after seeing the *entire* input.

Model

The model is similar to the model for Huffman coding.

Arithmetic coding based on an alphabet and the probability distribution of its symbols. Ideally, the probabilities are computed from the precise frequency counts of a source. This requires reading the entire source file before the encoding process. In practice, we can use an estimated or approximately fixed probability distribution with the cost of slightly lower compression ratio.

Similar to Huffman coding, the model can be a static or dynamic one depending on whether the probability distribution is changed during a coding process.

Coder

We introduce a simple version of Arithmetic coder here to give a flavour of how it works.

Suppose that both encoder and decoder know the length of the input sequence of symbols. Consider a simple case with a binary alphabet {A, B}, and the probability that the next input symbol is 'A' is p_A and p_B.

Encoding

Our essential goal is to assign an unique interval to each potential symbol sequence of a known length. After that, what we need to do is merely, from each interval, to select a suitable decimal number as the arithmetic code.

The Arithmetic encoder reads a source of sequence of symbols from the above binary alphabet one symbol at a time. Each time a new subinterval is derived depending on the probability of the input symbol. This process, starting with interval [0,1), is iterated until the end of input symbol sequence. Then the arithmetic coder outputs a chosen *decimal number* within the final subinterval for the entire input symbol sequence.

Example 7.1 *$p_A = 1/4$, and $p_B = 3/4$; and the symbol generation is 'memoryless'[2]. Show the ideas of the simple version of Arithmetic coding for an input sequence containing*

1. *a single symbol*
2. *two symbols.*

[2] *By memoryless, we mean that the probability distribution for generating any symbol at any moment is the same as that of its original lifetime distribution.*

Solution

1. Since p_A is 1/4, we first divide the interval [0,1) into two subintervals of which the size is proportional to the probability of each symbol, as shown in Figure 7.1. Using the convention that a real number $< 1/4$ represents A and a real number $\geq 1/4$ represents B. Let A be .0 (output 0) and B as .5 (output 5).

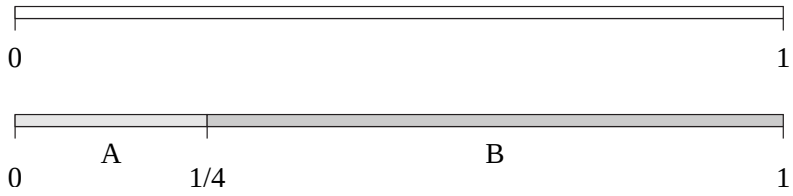


Figure 7.1: Encoding for one symbol.

2. Extending this to two symbols, we can see that $p_{AA} = p_A \times p_A = 1/16$, $p_{AB} = p_A \times p_B = 3/16$, $p_{BA} = p_B \times p_A = 3/16$, and $p_{BB} = p_B \times p_B = 9/16$. We divide each of the two subintervals further into two sub-subintervals of which the size is again proportional to the probability of each symbol, as show in Figure 7.2.

   We then encode as follows:

```
If the input is so far is 'AA', i.e. input a 'A' followed
by another 'A', then output a number
     <1/16=0.0625;
          (code .0, i.e. output 0)
If the input is 'AB' then output a number
     >=1/16=0.0625 but <1/16+3/16=1/4=0.25;
          (code .1, output 1)
If the input is 'BA' then output a number
     >=1/4=0.25 but <1/4+3/16=7/16=0.4375;
          (code .3, output 3)
```

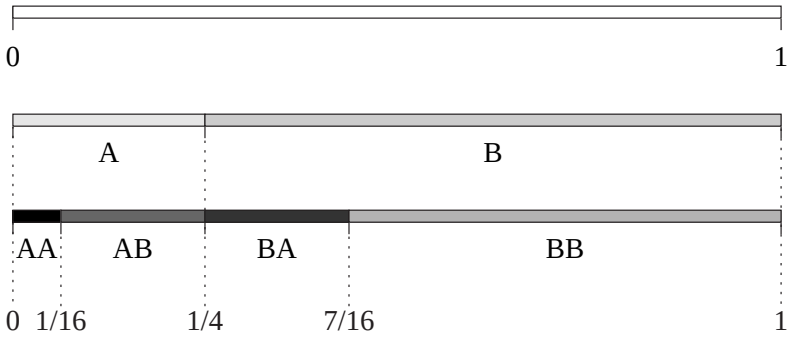


Figure 7.2: Encoding for two symbols.

```
If the input is 'BB' then output a number
     >=7/16=0.4375 but <7/16+9/16=1;
         (code .5, output 5).
```

Similarly, we can extend this to a sequence with 3 symbols: We can easily work out the probability distribution $\{p_{AAA}, p_{AAB}, ...p_{BBB}\}$ and therefore decide which subinterval for each possible sequence.

In theory, this subinterval division process can be repeated as many times as the number of input symbols requires.

Unique-decodability

We have shown in the previous section how it is possible to divide an interval starting from [0.1) according to the probability of the input symbol each time and how to assign the final interval to every possible symbol sequence of a certain length.

As we can see from the example that, in theory, there is no overlap among these subintervals for all possible symbol sequences. It is this 'overlap-free' fact that makes arithmetic codes uniquely decodable.

Observation

- We could choose, say, 0.05123456789 (<0.625) to encode `AA` but we do not, because 0.0 allows a *one* bit code, instead of a 11 bit one (for 0.05123456789).
- There are other equally good codes, For example, `AB 0.2; BA 0.4; BB 0.6 (or 0.7, 0.8, 0.9)`
- The code for `A` is the same as the code for `AA`. This is acceptable because the encoder and the decoder are both assumed to know the length of the input sequence.

Encoding main steps

From previous examples, we know that the encoding process is essentially a process to derive the final interval following the main steps below:

1. Let the initial interval be [0,1)
2. Repeat the following until the end of input sequence:

(a) read the next symbol `s` in the input sequence

(b) divide the current interval into subintervals whose sizes are proportional to the symbols probabilities

(c) make the subinterval for the sequence up to `s` the *new* current interval

3. (When the entire input sequence has been processed in this way), output a decimal number within the current interval.

Defining an interval

There are usually two ways to describe an interval:

1. $[L, H]$, where variable L stores the lowest value of the interval range and H stores the highest.
2. $[L, L + \delta]$, where variable L stores the lowest value of the interval range and δ stores the distance between the highest and the lowest value of the interval.

We use the later approach for programming convenience.

Encoding algorithm

We first look at a simple case for a small alphabet containing only two symbols.

Let the current interval be $[L, L + \delta)$ at each stage, and the output fraction x satisfy $L \leq x < L + \delta$.

Initially, L=0 and $\delta = 1$, this gives [0,1). If the next symbol could either be s_1 or s_2 with probability
p_1 and p_2 respectively, then we assign the intervals

$$[L, \underline{L + \delta p_1}) \text{ and } [\underline{L + \delta p_1}, \underline{L + \delta p_1} + \delta p_2)$$

and select the appropriate one. Note: $L + \delta p_1 + \delta p_2 = L + \delta(p_1 + p_2) = L + \delta$

Let `d` be δ in the algorithm below[3].

[3] *In the algorithms below, '$<-$' is used as an assignment operation (e.g.* `L` $<-$ `0` *means* `L` *is assigned a value* `0`*)*

```
1. /* Initialise */ L <- 0 and d <- 1
2. Read next symbol
3. /* Update intervals */
   If next symbol is s1,
     then leave L unchanged, and set d=d*p1
     else L <- L + d*p1 and d <- d*p2
4. If no more symbols left
     then output fraction from [L, L+d)
     else go to step 2.
```

Example 7.2 *For the example earlier, let $s_1 = A$ and $s_2 = B$ with probabilities $p_1 = 1/4$ and $p_2 = 3/4$. We encode ABA as follows:*

- *$L = 0$ and $d = 1$*

- *read symbol A, leave* $L = 0$ *and set* $d = 1/4$
- *read symbol B, set* $L = 1/16$ *and* $d = 3/16$
- *read symbol A, leave* $L = 1/16$ *and set* $d = 3/64$
- *Done, so choose a decimal number* $\geq L = 1/16 = 0.0625$ *but* $< L + \delta = 0.109375$.
  For example, choose 0.1 and output 1.

The following table shows arithmetic encoding algorithm that can be applied to encode all length 3 strings over the alphabet {A,B} with $p_A = 1/4$ and $p_B = 3/4$.

Seq	Prob	Interval (Fraction)	Interval (Decimal)	Output	$log_{10}p$
AAA	1/64	[0, 1/64)	[0, .015625)	0	1.81
AAB	3/64	[1/64,4/64)	[.015625, .0625)	.02	1.33
ABA	3/64	[4/64,7/64)	[.0625,.109375)	.1	1.33
ABB	9/64	[7/64,16/64)	[.109375, .25)	.2	.85
BAA	3/64	[16/64,19/64)	[.25,.296875)	.25	1.33
BAB	9/64	[19/64,28/64)	[.296875,.4375)	.3	.85
BBA	9/64	[28/64,37/64)	[.4375,.578125)	.5	.85
BBB	27/64	[37/64,1)	[.578125, 1)	.6	.37

Observation

In the algorithm and examples that we have discussed so far:

- A decimal system (base 10) is used for (description) convenience.
- The average length of the code is

  $1/64 + 6/64 + 3/64 + 9/64 + 6/64 + 9/64 + 9/64 + 27/64 = 70/64 = 1.09$ digits

- The information theoretic bound, i.e. the entropy, is

  $$0.244 \times 3 = 0.73 \text{ digits.}$$

- Huffman coding would require 3 digits for this case.

Decoding

The decoding process is the inverse of the encoding process.

Let p1 be the probability for s1 and p2 be the probability for s2. Given a string of digits which is a real number x in [0, 1), ($d = \delta$):

```
1. /* Initialise */ L <- 0 and d <- 1
2. Read x,
3. /* Computer the interval */
   If x is a member of [L,L+d*p1)
      then output s1, leave L unchanged, and
          set d<-d*p1
      else /* x is a member of [L+d*p1, L+d) */
          then output s2, set L<- L+d*p1 and d<-d*p2.
4. If the_number_of_decoded_symbols
              < the_required_number_of_symbols
      then go to step 2.
```

Renormalisation

Note the number of x's digits in the algorithms earlier can be quite large. On most computers, the value of δ would become zero rapidly (say after 300 symbols or so).

This is a technique for dealing with an implementation problem. The idea is to **stop** δ and L becoming zero rapidly by resetting the intervals.

Example 7.3 *Consider the Arithmetic encoding for* `baabaa`.

After `baa` *the interval is [.25,.296875), and we know that the first digit of the output must be 25. So output 25 and reset the interval to [.5,.96875); both become two to three times as big. This stops δ and L going to zero.*

Coding a larger alphabet

Arithmetic coding can be generalised to larger alphabets.

Example 7.4 *Given 3 symbols A,B,C with probabilities 0.5, 0.3 and 0.2, the current allowed interval would be subdivided into 3 intervals according to the ratio 5:3:2. We would choose the new allowed interval among the three.*

Effectiveness

The Arithmetic coding algorithms codes a sequence with probability p with at most $\lceil -\log_{10} p \rceil$ digits, and this is as close to the optimal as we will get in a digital world.

Learning outcomes

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

- explain the main ideas of Arithmetic coding
- describe, with an example of a small alphabet {A,B}, how encoding and decoding Arithmetic algorithms work
- discuss the advantages of Arithmetic coding compared with Huffman coding
- explain the main problems in implementation of Arithmetic coding.

Activities

1. What are the advantages of Arithmetic coding compared with the problems of Huffman coding? Use the following simple version of the encoding algorithm in the discussion.

```
1. /* Initialise */ L <- 0 and d <- 1
2. Read next symbol
3. /* Update intervals */
   If next symbol is s1,
     then leave L unchanged, and set d=d*p1
     else L <- L + d*p1 and d <- d*p2
4. If no more symbols left
     then output fraction from [L, L+d)
     else go to step 2.
```

2. Demonstrate how to encode a sequence of 5 symbols, namely `BABAB` from the alphabet {A,B} using the Arithmetic coding algorithm if $p_A = 1/5$ and $p_B = 4/5$.

3. Using the coding in the previous question as an example, explain how the Arithmetic decoding algorithm works. For example, explaining how to get the original `BABAB` back from the compressed result.

4. Show how to encode a sequence of 5 symbols from {A, B}, namely ABABB using arithmetic coding if $p_A = 1/5$ and $p_B = 4/5$. Would it be possible to have an coded output derived from a decimal value .24? Describe how the decoding algorithm works.

5. A sequence of 4 symbols from {A,B} was encoded using arithmetic coding. Assume $p_A = 1/5$ and $p_B = 4/5$. If the coded output is 0.24, derive the decoded output step by step.

Laboratory

1. Implement a simple version of the Arithmetic encoding algorithm for a binary alphabet, e.g (A,B).

2. Implement a simple version of the Arithmetic decoding algorithm for the same binary alphabet (A,B).

Sample examination questions

1. Describe briefly how Arithmetic Coding gets around the problems of Huffman Coding. You may use the following simple version of the encoding algorithm for discussion.

```
1. /* Initialise */ L <- 0 and d <- 1
2. Read next symbol
3. /* Update intervals */
   If next symbol is s1,
     then leave L unchanged, and set d=d*p1
     else L <- L + d*p1 and d <- d*p2
4. If no more symbols left
     then output fraction from [L, L+d)
     else go to step 2.
```

2. Show how the Arithmetic coding is applied to code all length-2 strings over the alphabet {A, B}. Suppose $p_A = 1/4$ and $p_B = 3/4$. You may like to summarise the code in the following format:

Sequence	Probability	Interval Fraction	Interval Decimal	Output
???	???	???	???	???
...				

Chapter 8

Dictionary based compression

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 3.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 2.

In this chapter, we look at a set of algorithms based on a dictionary model.

Dictionary based compression

These are techniques that incorporate the structure in the data in order to achieve a better compression. The main idea is to eliminate the redundancy of storing repetitive strings for words and phrases repeated within the text stream. The coder keeps a record of the most common words or phrases in a document and use their indices in the dictionary as output tokens. Ideally, the tokens are much shorter in comparison with the words or phrases themselves and the words and phrases are frequently repeated in the document.

A list of commonly occurring patterns are built within the coder, including:

- front-end: this involves the selection of strings to be coded
- dictionary: this is used to match the strings to short codewords, such as indices, small integers or small pointers.

Both compressors and decompressors have to maintain a dictionary. The dictionary-based algorithms are normally much faster than entropy-based ones, for they read the input as characters rather than as streams of bits.

Dictionary based approaches can be

- adaptive[1]
- *variable-to-fixed* in basic form
- fixed-to-variable in implementation
- variable-to-variable overall.

[1] *Note that there can be static dictionaries.*

Applications include UNIX `compress`.

Popular algorithms

The most popular dictionary compression algorithms are:

- LZ77 (also known as *LZ1*)
- LZ78 (*LZ2*)
- LZW

The algorithms are named after the authors Abraham Lempel and Jakob Ziv who published the papers in 1977 and 1978. A popular variant of LZ78, known as (basic) *LZW* was published by Terry Welch in 1984. There are numerous variants of LZ77 and LZ78/LZW. We only, however, outline each of these algorithms here.

LZW coding

People often found that LZW algorithms are easier to understand and are the most popular ones. We hence study LZW coding first.

Encoding

The dictionary usually contains 256 entries (e.g.ASCII codes) of single characters initially[2].

[2] *In the algorithms below, words between '/*' and '*/' signs are comments.*

The encoding algorithm is:

```
1. word='';
2. while not end_of_file
      x=read_next_character;
      if word+x is in the dictionary
                  /* where + means concatenate */
         word=word+x
      else
         output the dictionary index for word;
         add word+x to the dictionary;
         word=x;
3. /* now end_of_file is true */
   output the dictionary index for word;
```

Example 8.1 *Trace the operations of the LZW algorithm on the input string* ACBBAAC.

Suppose the first 256 places in the dictionary have been filled initially with symbols as below:

```
Dictionary:
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
        A B C D E F G H I J  K  L  M  N  O  P  Q  R  S  T

21 22 23 24 25 26 27     ...    256
U  V  W  X  Y  Z  space  ...    9
```

Solution

1. Initial step:

   ```
   word: ''
   ```

2. Symbols to read: ACBBAAC

   ```
   read next_character x: A (in dictionary)
   word+x: A
   word  : A
   ```

3. Symbols to read: CBBAAC

```
read next_character x: C
word+x: AC (not in dictionary)
output: 1
new entry of the dictionary:  257
                              AC
word  : C
```

4. Symbols to read: BBAAC

```
read next_character x: B
word+x: CB (not in dictionary)
output: 3
new entry of the dictionary:  258
                              CB
word  : B
```

5. Symbols to read: BAAC

```
read next_character x: B
word+x: BB (not in the dictionary)
output: 2
new entry of the dictionary:  259
                              BB
word  : B
```

6. Symbols to read: AAC

```
read next_character x: A
word+x: BA (not in the dictionary)
output: 2
new entry of the dictionary:  260
                              BA
word  : A
```

7. Symbols to read: AC

```
read next_character x: A
word+x: AA (not in the dictionary)
output: 1
new entry of the dictionary:  261
                              AA
word  : A
```

8. Symbols to read: C

```
read next_character x: C
word+x: AC (in dictionary)
word  : AC
```

9. Symbols to read: none

```
read next_character x: end_of_file
output: 257
```

So the total output (the compressed file) is: 1 3 2 2 1 257.

The new entries of the dictionary are:

```
257  258  259  260  261
AC   CB   BB   BA   AA
```

Example 8.2 *Encode* AAABAABBBB *by tracing the LZW algorithm.*

Solution

Suppose the first 256 places in the dictionary have been filled initially with symbols as below:

```
Dictionary:
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
        A B C D E F G H I J  K  L  M  N  O  P  Q  R  S  T

21 22 23 24 25 26 27     ...    256
U  V  W  X  Y  Z  space  ...    9
```

1. Initial step:

   ```
   word: ‘’
   ```

   We show the actions taken and the variable values updated by the encoding algorithm on completion of each iteration of step 2. as follows:

2. Symbols to read: AAABAABBBB

   ```
   Input next_character (x): A
   word+x: A
   word  : A
   ```

3. Symbols to read: AABAABBBB

   ```
   Input next_character (x): A
   word+x: AA
   Output token: 1
   Dictionary (new entries): 257
                             AA
   word  : A
   ```

4. Symbols to read: ABAABBBB

   ```
   Input next_character (x): A
   word+x: AA
   word  : AA
   ```

5. Symbols to read: BAABBBB

   ```
   Input next_character (x): B
   word+x: AAB
   Output token: 257
   Dictionary (new entries): 257 258
                             AA  AAB
   word  : B
   ```

6. Symbols to read: AABBBB

```
Input next_character (x): A
word+x: BA
Output token: 2
Dictionary (new entries): 257 258 259
                          AA  AAB BA
word  : A
```

7. Symbols to read: ABBBB

```
Input next_character (x): A
word+x: AA
word  : AA
```

8. Symbols to read: BBBB

```
Input next_character (x): B
word+x: AAB
word  : AAB
```

9. Symbols to read: BBB

```
Input next_character (x): B
word+x: AABB
Output token: 258
Dictionary (new entries): 257 258 259 260
                          AA  AAB BA  AABB
word  : B
```

10. Symbols to read: BB

```
Input next_character (x): B
word+x: BB
Output token: 2
Dictionary (new entries): 257 258 259 260  261
                          AA  AAA BA  AABB BB
word  : B
```

11. Symbols to read: B

```
Input next_character (x): B
word+x: BB
word  : BB
```

12. Symbols to read: (none)

```
Input next_character (x): end_of_file
/* goes to step 3. */
Output token: 261
```

So the compressed output for AAABAABBBB is

```
1 257 2 258 2 261
```

The new entries of the dictionary are:

```
257 258 259 260  261
AA  AAB BA  AABB BB
```

Decoding

Similarly, the decoder builds its own dictionary as it reads the tokens one by one from the compressed file. Note: I have made some corrections to the decoding algorithm in book Wayner(2000) and use `element` to mean the dictionary entry at index (token) x.

The algorithm now is:

```
1. read a token x from the compressed file;
2. look up dictionary for element at x;
   output element
   word=element;
3. while not end_of_compressed_file do
      read x;
      look up dictionary for element at x;
      if there is no entry yet for index x
         then element=word+first_char_of_word;
      output element;
      add word+first_char_of_element to the dictionary;
      word=element;
4. end
```

Example 8.3 *Show the operations of the decompress algorithm on the input tokens* 1 3 2 2 1 257[3].

[3] *This is the compressed result from Example 8.1.*

Solution

1. (Initial step) Tokens to be read: 1 3 2 2 1 257

   ```
   read token x: 1
   element: A
   output: A
   word : A
   ```

2. Tokens to be read: 3 2 2 1 257

   ```
   read token x: 3
   element: C
   output: C
   add new entry of dictionary: 257
                                AC
   word : C
   ```

3. Tokens to be read: 2 2 1 257

```
read token x: 2
element: B
output: B
add new entry of dictionary: 258
                             CB
word : B
```

4. Tokens to be read: 2 1 257

```
read token x: 2
element: B
output: B
add new entry of dictionary: 259
                             BB
word : B
```

5. Tokens to be read: 1 257

```
read token x: 1
element: A
output A
add new entry of dictionary: 260
                             BA
word : A
```

6. Tokens to be read: 257

```
read token x: 257
element: AC
output AC
add new entry of dictionary: 261
                             AA
word : AC
```

7. Token to be read: (none)

```
read token x: end_of_compressed_file
(end)
```

So the total output is `ACBBAAC`.

The new entries of the dictionary are:

```
257  258  259  260  261
AC   CB   BB   BA   AA
```

Example 8.4 *Decode* 1 257 2 258 2 261[4] *of by tracing the decording algorithm.*

[4] *This is the compressed result from Example 8.2.*

Suppose again the first 256 places in the dictionary have been filled initially with symbols as below:

```
Dictionary:
        1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
        A B C D E F G H I J  K  L  M  N  O  P  Q  R  S  T

21 22 23 24 25 26 27     ...    256
U  V  W  X  Y  Z  space  ...    9
```

Solution

1. Tokens to be read: 1 257 2 258 2 261

   Initial steps:

   ```
   Read next_token (x): 1
   output: A
   word: A
   ```

2. Tokens to be read: 257 2 258 2 261

   (Iterations begin):

   ```
   Read next_token (x): 257
   Look up the dictionary and find there is no entry yet for 257
   element: AA /* element=word+first_char_of_word */
   output element: AA
   Dictionary (new entries so far): 257
                                    AA
   word: AA
   ```

3. Tokens to read: 2 258 2 261

   ```
   Read next_token (x): 2
   Look up the dictionary and output element: B
   Dictionary (new entries so far): 257 258
                                    AA  AAB
   word: B
   ```

4. Tokens to read: 258 2 261

   ```
   Read next_token (x): 258
   Look up the dictionary and output element: AAB
   Dictionary (new entries so far): 257 258 259
                                    AA  AAB BA
   word: AAB
   ```

5. Tokens to read: 2 261

   ```
   Read next_token (x): 2
   Look up the dictionary and output element: B
   Dictionary (new entries so far): 257 258 259 260
                                    AA  AAB BA  AABB
   word: B
   ```

6. Tokens to read: 261

```
Read next_token (x): 261
Look up the dictionary and find no entry for 261
element: BB /* element=word+first_char_of_word */
output element: BB
Dictionary (new entries so far): 257 258 259 260  261
                                 AA  AAB BA  AABB BB
word: BB
```

7. Tokens to read: (none)

```
Read next_token (x): end_of_compressed_file
(end)
```

So the decoded message is `AAABAABBBB`.

Dictionary (new entries) is :

```
257 258 259 260  261
AA  AAB BA  AABB BB
```

Observations

1. The compression algorithm and the decompression algorithm build an identical dictionary independently. The advantage of this is that the compression algorithm does not have to pass the dictionary to the decompressor.
2. The decompressor lags one symbol behind. As we can see from Example 8.1, it only adds `AA` to cell 261 after actually outputting the entire decompressed string `ACBBAAC`.

LZ77 family

In this method, the dictionary to use is a *portion* of the previously seen input file. A sliding window that can be shifted from left to right is maintained to define the dictionary part and to scan the input sequence of symbols.

The window consists of two parts:

- History buffer (also known as the Search buffer); this contains a portion of the recently seen symbol sequence
- Lookahead buffer; this contains the next portion of the sequence to be encoded.

The size of each buffer is fixed in advance. Let H be the size of the Search buffer and L be that of the Lookahead buffer.

Figure 8.1 shows an example of this setting: a History buffer of size $H = 16$ bytes (1 byte for each symbol) and Lookahead buffer of size $L = 12$, where the sequence of symbols in (and before) the History buffer has been seen and compressed but the sequence in (and after) the Lookahead buffer is to be compressed. You can image that the window moves from left to right (or the

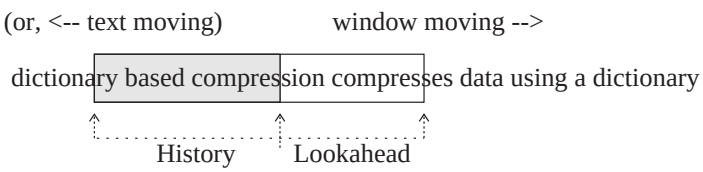


Figure 8.1: History buffer and Lookahead buffer

text sequence moves from right to left) from time to time during the compression process.

In practical implementation, the History buffer is some thousands of bytes long and the lookahead buffer is only tens of bytes long (Salomon 2002).

A typical compression step

Suppose that the History buffer has H bytes (characters) that have been seen and encoded; The Lookahead buffer has at most L characters, which have been seen but not yet encoded.

1. Read characters from the input until the Lookahead buffer is full
2. Scan the History buffer from right to left searching for a match to some prefix (of the sequence of symbols)[5] in the Lookahead buffer.
   If more than one match is found in the History buffer, take the longest and the first one from the right, i.e. the one located furthest on the right in the History buffer.
3. If a match of length $k \geq 2$ characters is found (see Figure 8.2) with an offset of j bytes[6] then output the pointer (j, k).
   Slide the History buffer window k characters to the right (i.e. move the first k characters in the Lookahead buffer into the end of the History buffer, and remove the leftmost k characters from History buffer).
4. If no match is found, or if the match is only 1 character long (see Figure 8.1), then output (0, ASCII(c)), where c is the next character in the Lookahead buffer.
   Slide History window 1 character to the right.

[5] *By prefix of a symbol sequence, we mean a subsequence of any length beginning with the first symbol of the sequence.*

[6] *By offset of j bytes, we mean the number of the symbols from the first matched symbol to the end of the History buffer inclusive is j.*

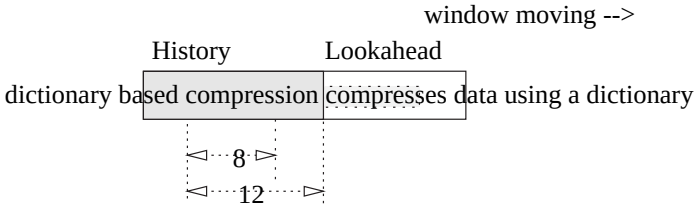


Figure 8.2: A match is found with an offset length 12 and match length 8.

Example 8.5 *Starting with a History buffer of size 16 bytes which contains the characters* `ased␣compression` *(where ␣ represents a space) and an empty Lookahead buffer, show the output and the final state of History buffer if the following characters are next in the input, for Lookahead buffer of size 12:* `␣compresses␣data␣using␣a␣dictionary`.

Solution We refer to Figure 8.2 and trace what happens step by step following the LZ77 encoding algorithm.

1. Read input string into Lookahead buffer and the History (left) and Lookahead (right) buffer now contain the following:

ased␣compression	␣compresses␣	data␣using␣a␣dictionary

   A match '␣compress' is found and we record the offset 12 (bytes) and the match length 9 (bytes).

   Output: (12,9).

   Slide the window 9 characters to the right: The History and Lookahead buffer now contain:

ression␣compress	es␣data␣usin	g␣a␣dictionary

2. A match 'es' is found[7] so we record the offset 3 and the match length 2.

   Output: (3,2).

   Slide the window 2 characters to the right:

   The History and Lookahead buffer now contain:

ssion␣compresses	␣data␣using␣	a␣dictionary

3. Only a one-symbol match '␣' is found so:

   Output: the pointer (0,ASCII('␣')).

   Move the window 1 place to the right:

   The History and Lookahead buffer now contain:

sion␣compresses␣	data␣using␣a	␣dictionary

4. No match is found so:

   Output: the pointer (0,ASCII('d')).

   Move the window 1 place to the right:

   The History and Lookahead buffer now contain:

ion␣compresses␣d	ata␣using␣a␣	dictionary

5. No match is found so

   Output the pointer (0, ASCII('a')).

   Move History 1 place to the right. The History (left) and Lookahead (right) buffer now contain:

on␣compresses␣da	ta␣using␣a␣d	ictionary

6. No match is found so

   Output: the pointer (0,ASCII('t')), and move History 1 place to the right.

   The History (left) and Lookahead (right) buffer now contain:

n␣compresses␣dat	a␣using␣a␣di	ctionary

7. Only one symbol match 'a' is found so

   Output the pointer (0, ASCII('a')), and move History 1 place to the right.

   The History (left) and Lookahead (right) buffer now contain:

␣compresses␣data	␣using␣a␣dic	tionary

⋮

and so on ...

[7] *In fact, 'es' is found in 2 places but we only consider the first (rightest) match here.*

The decompression algorithm

Decoder also maintains a buffer of the same size as the encoder's window. However, it is much simpler than the encoder because there is no matching problems to deal with. The decoder reads a token and decide whether the token represents a match.

If it is a match, the decoder will use the offset and the match length in the token to reconstruct the match. Otherwise, it outputs the ASCII code in the token.

We use an example to show how this works.

Example 8.6 *Suppose the buffer contains the following decoded symbols:* `Dictionary based␣compression`.

The decoder reads the following tokens one at each iteration: `(12,9) (3,2) (0,ASCII(‘␣’)) (0,ASCII(‘d’)) (0, ASCII(‘a’)) (0,ASCII(‘t’)) (0, ASCII(‘a’))`.

Solution

1. Read the next token `(12,9)`, which is a match. The decoder uses the offset (the first number in the token) to find the first symbol of the match and uses the match length (the second number in the token) to decide the number of symbols to copy.

   So we have the following process:

   (a) Count 12 from the end of the buffer and find the symbol '␣'

   (b) Copy the next 9 symbols one by one (see the underlined characters) and update the buffer:

   `D ictionary based␣compression␣`
   `Di ctionary based␣c compression␣c`
   `Dic tionary based␣co mpression␣co`
   `Dict ionary based␣com pression␣com`
   `Dicti onary based␣comp ression␣comp`
   `Dictio nary based␣compr ession␣compr`
   `Diction ary based␣compre ssion␣compre`
   `Dictiona ry based␣compres sion␣compres`
   `Dictionar y based␣compress ion␣compress`

2. Read `(3,2)`, the buffer becomes:
   `based␣compression␣compresses`

3. Read `(0,ASCII(‘␣’))`, which is not a match and output '␣'.
   `sed␣compression␣compresses␣`

4. Read `(0,ASCII(‘d’))`, which is not a match so output 'd' (i.e. add 'd' into the buffer).
   `ed␣compression␣compresses␣d`

5. Read `(0,ASCII(‘a’))`, which is not a match so output 'a'.
   `d␣compression␣compresses␣da`

6. Read `(0,ASCII('t'))`, which is not a match so output 't'.
   `␣compression␣compresses␣dat`
7. Read `(0,ASCII('a'))`, which is not a match so output 'a'.
   `compression␣compresses␣data`

⋮

and so on.

Observations

- The History buffer is initialised to have no characters in it. Hence the first few characters are coded as raw ASCII codes (and the overhead) as the History buffer fills up. This may result in expansion rather than compression at the beginning.
- In common with the adaptive Huffman algorithm, LZ77 is an adaptive algorithm and may start with an empty model.
- The decompression algorithm builds up the same History buffer as the compression algorithm, and decodes pointers with reference to this History buffer. The decompression algorithm knows if the next character is a 'real' pointer or a raw character by looking at the first component of the pointer.
- In LZ77, there is an important design decision concerns the values of H and L to be made:
  1. Choosing a large History buffer means it is most likely that matches are found, but the offsets will be larger. A smaller buffer means smaller pointers but less chance of finding a match.
  2. Choosing a small Lookahead buffer means a quick search for a prefix, but the size of the match found will be limited.
- The algorithm is *asymmetric* since compression is slower than decompression. Compression involves searching for a match, which is computationally intensive, but decompression only involves reading out values from the History buffer.

LZ78 family

[8] *This is a commonly used data structure for strings and similar to trees.*

Typical LZ78 compression algorithms use a *trie*[8] to keep track all of the strings seen so far. The dictionary D contains a set of strings, which are numbered from 0 onwards using integers. The number corresponding to a string is called the token of the string. The output of the encoding algorithm is a sequence of tokens. Initially the dictionary D is normally loaded with all 256 single-character strings, and the token of each single-character string is simply its ASCII code. All subsequent strings are given token numbers 256 or more.

One compression step

Let `x` be the currently matched string. Initially `x` is empty.

The encoding algorithm is:

```
c:=next_char();
while (xc in D) do
(* xc == string x followed by character c*)
  begin
    x:=xc;
    c:=next_char();
  end
output token corresponding to x;
add xc to D, and give it
  the next available token
x:=c;
```

Applications

- UNIX utility `compress` is a widely used LZW variant.
  - The number of bits used for representing tokens is increased as needed gradually.
    For example, when the token number reaches 255, all the tokens are coded using 9 bits, until the token number 511 ($2^9 - 1$) is reached. After that, 10 bits are used to code tokens and so on.
  - When the dictionary is full (i.e. the token number reaches the limit), the algorithm stops adapting and only uses existing strings in the dictionary. At this time, the compression performance is monitored, and the dictionary is *rebuilt* from scratch if the performance deteriorates significantly.
  - The dictionary is represented using a *trie* data structure.
- GIF (Graphics Interchange Format) is a lossless image compression format introduced by CompuServe in 1987.
  - Each pixel of the images is an index into a table that specifies a colour map.
  - The colour table is allowed to be specified along with each image (or with a group of images sharing the map).
  - The table forms an uncompressed prefix to the image file, and may specify up to 256 colour table entries each of 24 bits.
  - The image is really a sequence of 256 different symbols and is compressed using the LZW algorithm.
- V.42bis compression is the ITU-T standard for data compression for telephone-line modems.
  - Each modem has a pair of dictionaries, one for incoming data and one for outgoing data.
  - The maximum dictionary size is negotiated between the calling and called modem as the connection is made. The minimum size is 512 tokens with a maximum of 6 characters per token.
  - Those tokens that are used infrequently may be deleted from the dictionary.
  - The modem may switch to transmitting uncompressed data if it detects that compression is not happening (e.g. if the file to transmit has already been compressed).
  - The modem may also request that the called modem discards the dictionary, when a new file is to be transmitted.

Highlight characteristics

We have studied all the important compression algorithms for compressing text. Some of them are static and others are adaptive. Some use fixed length

codes, others use variable length codes. The compression algorithms and decompression algorithms use the same model, others use different ones. We summarised these characteristics below:

Algorithm	Adaptive	Symmetric	Type
Run-length	n	y	variable to fixed
Huffman	n	y	fixed to variable
Adaptive Huffman	y	y	fixed to variable
Arithmetic	y	y	variable to variable
LZ77	y	n	variable to fixed
LZW	y	y	variable to fixed

Learning outcomes

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

- explain the main ideas of dictionary based compression
- describe compression and decompression algorithms such as LZ77, LZW and LZ78
- list and comment on the main implementation issues for dictionary based compression algorithms.

Activities

1. Using string `abbaacabbabbb#` (where `#` represents the end of the string) as an example, trace the values of word, x and the dictionary in running the basic Lempel-Ziv-Welch *encoding* and *decoding* algorithms, where word is the accumulated string and x is the character read in on each round.

2. Suppose the input for encoding is a string `aabbaaaccdee`. Demonstrate how the simplified version algorithms mentioned above work step by step.

   Suppose the History buffer is 7 characters long and the LookAHead buffer is 4 characters long. Illustrate how LZ77 compression and decompression algorithms work by analysing the buffer content on each main step.

   For LZ78 and LZW, show what the dictionaries built are look like at the completion of the input sequence in the following format.

```
Dictionary                Dictionary
address (in decimal)      entry
-----------------------------------
0                         ?, ?
1                         ?, ?
2                         ?, ?
3                         ?, ?
...
```

3. Describe a simplified version of LZ77, LZ78 and LZW algorithms. Analyse the type of the simplified algorithms in terms of, for example, whether they are *static or adaptive* and *fixed-to-variable*, *variable-to-fixed* or *variable-to-variable*.

Laboratory

1. Implement a simple version of LZW encoding algorithm.

2. Implement a simple version of LZW decoding algorithm.

Sample examination questions

1. Explain why a Dictionary-based coding method such as LZ77 is said to be *adaptive* and to be *variable-to-fixed* in its basic form.

2. One simplified version of LZ78/LZW algorithm can be described as below, where n is a pointer to another location in the dictionary and c is a symbol drawn from the source alphabet, and < n,c > can form a node of a linked list. Suppose that the pointer variable n also serves as the transmitted code word, which consists of 8 bits. Suppose also the 0 address entry in the dictionary is a NULL symbol.

   The dictionary is initialised by constructing

```
/* A Linked-list LZ78/LZW algorithm */
---------------------------------------
1. n:=0; Fetch next source symbol c;
2. If the ordered pair <n,c> is already in the dictionary
   then n:= dictionary address of entry <n,c>;
   else
```

```
     begin
       transmit n /* as a code word to decoder ??? */
       create new dictionary entry <n,c> at dictionary address m;
       m:=m+1;
       n:= dictionary address of entry <0,c>;
     end;
3. Return to Step 1.
```

Suppose that a *binary* information source emits the sequence of symbols 11000101100101110001111 (That is 110 001 011 001 011 100 011 11 - without spaces). Construct the encoding dictionary step by step and show, in the format below, the dictionary on completion of the above input sequence.

Dictionary address (in decimal)	Dictionary entry
0	?, ?
1	?, ?
2	?, ?
3	?, ?
...	

Chapter 9

Image data

Essential reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 3.1-3.2.

Further reading

Chapman, Nigel and Chapman, Jenny *Digital Multimedia.* (John Wiley & Sons, 2000) [ISBN 0-471-98386-1]. Chapter 3-6.

The compression techniques we have discussed so far are essentially developed for text compression. In this chapter, we look at a different type of data, namely, digital images.

Image data is special in the area of data compression in comparison to video and audio data because it was the first challenge to modern data compression where much larger files have to be processed often as routine, and it was the first type of data to which *lossy* compression methods can be applied.

Why do image files tend to be large compared with text? Why does image data allow lossy compression to be used? We look at the digital images in order to answer these questions and the answers are important for us to understand image compression discussed in next chapter.

In the multimedia world today, two types of images are most commonly used: one is the so-called *bitmap images*[1] and the other is *vector graphics*.

[1] *i.e. bitmap images.*

Bitmap images

These are also called *photographic* images for 2-dimensional pictures. A bitmap image is an array of pixel values. The data are the values of the pixels. Many bitmap images are created from digital devices such as scanners or digital cameras, or from programs such Painter which allows visual artists to paint images.

Resolution

A bitmap image is often measured by *resolution* which is a measure of how finely a device approximates continuous images using finite number of pixels. This is closely related to sampling rates.

There are two common ways of describing resolution:

- **dpi** stands for 'dots per inch' and represents the number of dots per unit length for data from devices such as scanners.
- **pixel dimension** is the number of pixels per frame for the video data from digital cameras.

For TV systems, the PAL frame is 768 by 576 pixels and the NTSC frame is 640 by 480. For computer monitors, the common resolutions are 640 by 480 or 1024 by 768.

Displaying bitmap images

Since a bitmap image is in fact represented in a computer by an array of pixel values, there has to be a way of mapping each pixel value to the physical dots on a display screen.

In the simplest case, each pixel corresponds one-to-one to the dots on the screen. In the general case, the physical size of the display of an image will depend on the device resolution. The pixel values are stored at different resolutions from the displayed image. Computations for scaling and clipping usually need to be performed to display a bitmap image.

In general, we have the following formula to decide the image sign and scaling:

```
imageDimension = pixelDimension/deviceResolution.
```

where the `deviceResolution` is measured in dpi.

Vector graphics

The image in vector graphics is stored as a mathematical description of a collection of graphic components such as lines, curves, and shapes and so on.

Vector graphics are normally more compact, scalable, resolution independent and easy to edit. They are very good for 3-D models on computer, usually built up using shapes or geometric objects that can easily be described mathematically.

Storing graphic components

The graphic components are stored by their mathematical functions instead of pixels.

For example, consider storing a line in the computer.

Example 9.1 *The mathematical function for a straight line is $y = kx + b$, where k is the slope and b the intercept. The function, instead of all the pixels, can be stored for the line.*

Since a geometric line is actually a finite segment of a line, we also need to store the two end points of the segment. In fact, it is sufficient to only store the coordinates of the two end points because we know the line function can be represented by two point coordinates. That is, given (x_1, y_1) and (x_2, y_2), we can easily write the function for the line:
$y = (y_2 - y_1)/(x_2 - x_1)x + (x_2y_1 - x_1y_2)/(x_2 - x_1)$.

Displaying vector graphic images

This would require some computation to be performed in order to interpret the data and generate an array of pixels to be displayed. The process of generating a pattern of pixels from a model is called *rendering*.

If the image is a line, only two end points are stored. When the model is rendered for display, a pattern of pixels has to be generated to display the line with the two end points.

Difference between vector and bitmap graphics

The differences between the two can be very obvious in terms of visual characteristics, but we are concerned more about the following issues:

1. The **requirements** of the computer system: a bitmap image must record the value of every pixel, but vector description takes much less space, so is more economical.
2. The **size** of bitmap images depends on the size of the image and the resolution but is independent of the complex of the image; while the size of vector graphics depends on the number of objects of which the image consists, it is independent of any resolution.
3. The **approach** of so-called *painting* programs produces bitmap images while that of *drawing* programs produces vector graphics. Very few packages offer support to both types of images.
4. The **behaviour** of the bitmap images and that of vector graphics are different when re-sized or scaled.

Combining vector graphics and bitmap images

The combination requires a transformation between vector graphics and bitmap images.

Rasterising

A raster is a predetermined pattern of scanning lines to provide substantially uniform coverage of a display area.

Rasterising is the process of interpreting the vector description of graphics for a display area. . A vector graphic loses all the vector properties during the process, e.g. the individual shapes can no longer be selected or moved because they become pixels.

Vectorisation

This is a more complicated process of transforming pixels to vectors. The difficulties arise from the need to identify the boundary of a pixel image using the available forts of curves and lines and to colour them. The vector file tends to be much bigger than the pixels file before the vectorization.

Colour

As we know, colour is a subjective sensation experience of the human visual system and the brain. It is a complicated system. Nevertheless, according to simplified so-called 'tristimulus' theory, in practice we can define a particular colour by giving the proportions of Red, Green and Blue light that the colour contains.

RGB colour model

This is a model for computing purposes. It comes from the idea of constructing a colour out of so-called *additive primary colours* (i.e. Red, Green and Blue or the RGB for short). Although there is no universally accepted standard for RGB, television and video industry does have a standard version of RGB colour derived in the Recommendation ITU-R BT.709 for High Definition TV (HDTV). Monitors are increasingly being built to follow the recommendation.

In the RGB model, we assume that all colours can in principle be represented as combinations of certain amounts of red, green and blue. Here the amount means the proportion of some standard of primary red, green and blue.

Example 9.2 *Suppose that the proportion is represented as percentages. (100%,0%,0%) then represents a colour of 'pure' red, and*

- *(50%,0%,0%) a 'darker' red*
- *(0%, 0%, 100%) a 'pure' blue*
- *(0%, 0%, 0%) black*
- *(100%, 100%, 100%)* ⋮*and so on.*

In fact, there are two commonly used representations of digital colours: they are so-called *RGB representation* and *LC representation*.

RGB representation and colour depth

Since it is only the relative values of R, G and B that matters, we can actually choose any convenient value range, as long as the range provided distinguishes enough different values.

In a common RGB representation, we use 1 byte (8 bits) to describe the lightness of each of the three primary colours, which gives a range of [0,255] (or [1,256]). Three bytes ($8 \times 3 = 24$ bits) describe each pixel of a colour image. In this way, there are 256^3 (i.e. 16,777,216) different colours can be represented in this way.

The number of bits used to hold a colour value is often called *colour depth*. For example, if we use 24 bits to represent one pixel, then the colour depth is 24. We sometimes also say '24 bit colour' to refer to a colour with 24 bit colour depth.

Example 9.3 *Some colours from the Linux colour database are 24 bit colours:*

```
R   G   B     colour          G   B     colour
------------------------------------------------------
255 255 255   white           255 250 250   snow
0   0   128   navy blue       248 248 255   ghost white
0   0   255   blue1           255 239 213   papaya whip
0   255 0     green           255 228 225   misty rose
...
```

LC representation

This is another common representation which is based on *luminance* (Y) and *chrominance* (C) values. Luminance Y reflects the brightness, and by itself gives a gray-scale version of the image.

The approach is based on colour differences. The idea is to separate the brightness information of an image from its colour. By separating brightness and colour, it is possible to transmit a picture in a way that the colour information is undetected by a black and white receiver, which can simply treat the brightness as a monochrome signal.

A formula is used which has been empirically determined for the best gray-scale likeness of a colour image.

$$Y = 0.299R + 0.587G + 0.114B$$

The chrominance (colour) components provide the additional information needed to *convert* the gray-scale image to a colour image. These components are represented by two values C_b and C_r given by $C_b = B - Y$ and $C_r = R - Y$.

This technique was initially used during the development from black-and-white to colour TV. However, it turns out to have some advantages. For example, since the human eye is much better at distinguishing subtle differences in brightness than subtle differences in colour, we can compress the Y component with greater accuracy (lower compression ratio), and make up for it by compressing the C_b and C_r components with less accuracy (higher compression ratio).

Classifying images by colour

For compression purposes, we can classify the images for compression by number of colours used. Three approaches are available, namely *bi-level image*, *gray-scale image*, *colour image*.

Bi-level image

This is actually the image with 1 bit colour. One bit allows us to distinguish two different colours.

Images are captured by scanners using only *two* intensity levels, one is 'information' and the other 'background'. Applications include:

- Text, line drawings or illustrations
- FAX documents for transmission
- Sometimes photographs with shades of gray.

Gray-scale image

This is actually the image with a 8-bit colour depth. The images are captured by scanners using multiple intensity levels to record shadings between black and white.

We use 8 bits to represent one pixel, and hold one colour value to provide 256 different shades of grey.

Gray scale images are appropriate for medical images as well as black-and-white photos.

Colour image

Colour images are captured by scanners, using multiple intensity levels and filtering to capture the brightness levels for each of primary colours, R, G and B (Red, Green, Blue).

Some computer systems use 16 bits to hold colour values. In this case, either 1 bit is left unused or different numbers of bits are assigned to R, G and B. If it is the latter, then we usually assign 5 bits for R and B, but 6 bits for G. This allocation is due to the fact that the human eye is more sensitive to green light than to red and blue.

Most common colour images are of a 24 bits colour depth. Although 24 bits are sufficient to represent more colours than the eye can distinguish, higher colour depths such as 30, 36 or even 48 bits are increasingly used, especially by scanners.

Classifying images by appearance

Images can be classified by their appearances which are caused by the way in which colours distribute, namely *continuous-tone image*, *discrete-ton image* and *cartoon-like image*.

Continuous-tone image

This type of image is a relatively natural image which may contain areas with colours. The colours seem to vary continuously as the eye moves along the picture area. This is because the image usually has many similar colours or grayscales and the eye cannot easily distinguish the tiny changes of colour when the adjacent pixels differ, say, by one unit.

Examples of this type of image are the photographs taken by digital cameras or the photographs or paintings scanned in by scanners.

Discrete-tone image

Alternative names are *graphical image* or *synthetic image*. This type of image is a relatively artificial image. There is usually no noise and blurring as there is in a natural image. Adjacent pixels in a discrete-tone image are often either identical or vary significantly in value. It is possible for a discrete-tone image to contain many repeated characters or patterns.

Examples of this type of images are photographs of artificial objects, a page of text, a chart and the contents of a computer screen.

Cartoon-like image

This type of image may consist of uniform colour areas but adjacent areas may have very different colours. The uniform colour areas in cartoon-like images are regarded as good features for compression.

Colour depth and storage

Colour depth determines the size of the bitmap image: each pixel requires 24 bits for 24 bit colour, but just a single bit for 1 bit colour. Hence, if the colour depth is reduced from 24 to 8, the size of a bitmap image will decrease by a factor of 3 (ignoring any fixed-size housekeeping information).

Image formats

There are lot of image software available these days. They have been developed on various platforms and used different formats or standards. Today's most widely used image formats include:

- GIF (Graphics Interchange Format)
- JPEG (Joint Photographic Experts Group standard for compressing still images)[2]
- Animated (Graphics Interchange Format - Animated)
- BMP (Windows Bitmap format)
- EPS (Encapsulated PostScript file format)
- PNG (Portable Network Graphics)
- PSD (Adobe Photoshop's native file format)

[2] *We often use JPEG to mean the standard by this expert group, rather than the organisation itself.*

- PSD Layered (Adobe Photoshop's native file format)
- STN (MediaBin's proprietary STiNG file format)
- TIFF (Tagged Image File Format)
- TGA (TARGA bitmap graphics file format)

Observation

From the above discussion, we can see that:

1. Image files are usually large because an image is two-dimensional and can be displayed in so many colours. In a bitmap image, each pixel requires typically 24 bits to represent its colour.
2. The loss of some image features is totally acceptable as long as the Human visual system can tolerate. After all, an image exists only for people to view. It is this fact this makes lossy compression possible.
3. Each type of image contains a certain amount of redundancy but the cause of the redundancy varies, and this leads to different compression methods. That is why there have to be many different compression methods.

Learning outcomes

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

- illustrate how a picture can be represented by either a bitmap image or a vector image
- explain the main differences between bitmap and vector graphics in terms of visual characteristics
- illustrate RGB colour model and LC representation
- describe a few of the commonly used image formats.

Activities

1. If you consider the three components of an RGB colour to be Cartesian coordinates in a 3-dimensional space, and normalise them to lie between 0 and 1, you visualise RGB colour space as a unit cube, see Figure 9.1.

   (a) What colours correspond to the eight corners of this cube?

   (b) What does the straight line running from the origin to (1,1,1) shown in the figure represent?

   (c) Comment on the usefulness of this representation as a means of visualising colour.

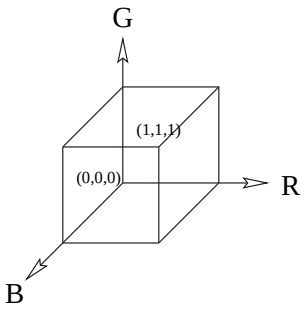


Figure 9.1: The RGB colour space

2. Classify and identify the following images for a continuous-tone image, discrete-tone image and cartoon-like image:

   - a reproduction of your national flag
   - a photograph of the surface of Mars
   - a photograph of yourself on the beach
   - a still from an old black and white movie.

3. Explain how RGB colour values, when $R = G = B$, can be used to represent shades of grey.

Laboratory

1. Investigate the colour utilities (or system, package, software) on your computer.

   For example, if you have access to Linux, you can go to `Start` → `Graphics` → `More Graphics` → `KColorChoser` and play with the colour display.

2. Experimenting on how the values of (R,G,B) and (H,S,V) affect the colour displayed.

Sample examination questions

1. Illustrate the RGB colour space using Cartesian coordinates in a 3-dimensional space.

2. Consider a square image to be displayed on screen. Suppose the image dimension is 20×20 square inches. The device resolution is 800 dpi. What would be the pixelDimension required? **Solution** 1600×1600

Chapter 10

Image compression

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 10.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 3.3-3.5, 3.7.

Image compression can be lossless or lossy, although most exist image compression algorithms are lossy. We shall introduce the general approaches in both classes for each type of image data discussed in the previous chapter.

However, one should bear in mind that:

1. In general, text compression is lossless by nature: there is no algorithm available to decide independently what text can be omitted.
2. In contrast, many image compression algorithms can delete a lot of image data effectively.
3. The successful performance of the lossy algorithms depends essentially on people's visual tolerance.

First we look at lossless image compression.

Lossless image compression

Lossless approaches mainly extend the compression techniques for text data to image compression.

Bi-level image

Examples include the fax standards: the ITU-T T.4 and T.6 recommendations.

Two popular approaches, *run-length coding* and *extended approach* are considered below:

Run-length coding

This approach (see previous chapter) can be applied to bi-level images due to the fact that:

1. each pixel in a bi-level image is represented by 1 bit with 2 states, say black (B) and white (W)
2. and the immediate neighbours of a pixel tend to be in an identical colour.

Therefore, the image can be scanned in row by row and the length of runs of black and white pixels can be computed. The lengths are then encoded by

variable-size codes and are written on the compressed file.

Example 10.1 *A character* **A** *represented by black-white pixels:*

```
  123456789012345678901234 5
1 BBBBBBBBBBBBBBBBBBBBBBBBB
2 BBBBBBBBBBBwwwBBBBBBBBBB
3 BBBBBBBBBBwwBwwBBBBBBBBB
4 BBBBBBBBBwwBBBwwBBBBBBBB
5 BBBBBBBBwwBBBBBwwBBBBBBB
6 BBBBBBBwwBBBBBBBwwBBBBBB
7 BBBBBBwwwwwwwwwwwwwBBBBB
8 BBBBBwwBBBBBBBBBBBwwBBBB
9 BBBBwwBBBBBBBBBBBBBwwBBB
0 BBBBBBBBBBBBBBBBBBBBBBBB
```

Solution We notice that Line 4 consists of several *runs* of pixels: 9B, 2w, 3B, 2w, 9B, i.e. 9 'B's followed by 2 'w's, followed by 3 'B's, then 2 'w's and then 9 'B's. Similarly, line 5 consists of 8B, 2w, 5B, 2w, 8B; ... and so on.

A better compression may be achieved by taking into consideration the *correlation* of adjacent pairs of lines.

For example, a simple idea is to code Line 5 with respect to Line 4 by the differences in number of pixels: -1, 0, 2, 0, -1.

Imagine if we send, via a telecommunication channel, the data `-1, 0, 2, 0, -1` for Line 5 after the data for Line 4. There would be no confusion nor need to send `7B, 2w, 5B, 2w, 7B` which includes more characters.

Note: The algorithm assumes that successive lines have the same number of runs. The actual ITU-T T.4 Group algorithm is much more complicated than this.

Extended approach

The idea of this approach extends the image compression principles and concludes that if the current pixel has colour B (or W) then black (or white) pixels seen in the past (or those that will be found in future) tend to have the same immediate neighbours.

The approach checks n of the near neighbours of the current pixel and assigns the neighbours an n-bit number. This number is called the *context* of the pixel. In principle there can be 2^n contexts, but the expected distribution is non-uniform because of the image's redundancy.

We then derive a probability distribution of the contexts by counting the occurrence of each context. For each pixel, the encoder can then use adaptive arithmetic coding to encode the pixel with the probabilities. This approach is actually used by JPEG.

Grayscale and colour images

The Graphics Interchange Format (GIF) was introduced by CompuServe in 1987.

In GIF, each pixel of the image is an index into a table that specifies a colour map for the entire image. There are only 256 different colours in the whole image. Of course, the colours may also be chosen from a predefined and much larger palette. GIF allows the colour table to be specified for each image, or for a group of images sharing the use of map or without a map at all.

Reflected Gray codes (RGC)

Reflected Gray codes (RGC) is a good representation for grayscale images. In this system, we assign codewords in such a way that any two consecutive numbers have codewords differing by 1 bit only.

Example 10.2 *We show below the decimal numbers that are represented by 1 bit, 2 bit and 3 bit RGC accordingly:*

```
decimal value:   0   1   2   3   4   5   6   7
1 bit RGC    :   0   1
2 bit RGC    :  00  01  11  10
3 bit RGC    : 000 001 011 010 110 111 101 100
... etc.
```

A RGC codeword can be derived from a normal binary codeword as follows:

Given a decimal number m, its Reflected Gray codeword is
$\boxed{m_2 \text{ XOR } \textit{shift-1-bit-to-right}(m_2)}$,
where m_2 represents the binary codeword of m.

Example 10.3 *Derive a 3-bit reflected Gray codeword for decimal 3.*

```
The 3-bit binary code of 3 is: 011
Shift 011 one bit to the right (and add 0 in front): 001
011 XOR 001 = 010

So the 3-bit reflected Gray codeword for 3 is 010.
```

Dividing a grayscale image

Using *Reflected Gray Code* (RGC), this approach separates a grayscale image into a number (say, n) of bi-level images and apply to each bi-level image a different compression algorithm depends on the characteristics of each bi-level image.

The idea is to assume intuitively that two *similar* adjacent pixels in the grayscale image are likely to be identical in most of the n bi-level images. By 'similar' in the examples below, we mean that the number of different bits between two codewords, i.e. the Hamming distance of two binary codewords, is small as well as the difference in value. For example, 0000 and 0001 are similar because their value difference is 1 and the number of different bit(s) is also 1. The two codewords are identical in the first three bits.

Now we look at an example of separating a grayscale image into n bi-level images.

Example 10.4 *Given a grayscale image with 8 shades of grey, we can represent each shade by 3 bits. Let each of the three bits, from left to right, be identified as the high, middle and low (bit).*

Suppose that part of a grayscale image is described by matrix `A` *below, where each RGC codeword represents a pixel with the shade of that value:*

```
      010  010  011  110
A =   001  011  010  111
      000  001  011  101
```

Then the image can be separated into 3 bi-level images (also called bitplanes) as follows:

1. *Bitplane* `A.high` *below consists of all the high bits of* `A`. `A.high` *can be obtained from* `A` *by removing, for each entry of* `A`*, the two bits other than the* `high` *bit.*

```
            0  0  0  1
A.high =    0  0  0  1
            0  0  0  1
```

2. `A.middle` *below consists of all the middle bits of* `A`. `A.middle` *can be obtained from* `A` *by removing, for each entry of* `A`*, the two bits other than the* `middle` *bit.*

```
              1  1  1  1
A.middle =    0  1  1  1
              0  0  1  0
```

3. `A.low` *below consists of all the low bits of* `A`. `A.low` *can be obtained from* `A` *by removing, for each entry of* `A`*, the two bits other than the* `low` *bit.*

```
            0  0  1  0
A.low =     1  1  0  1
            0  1  1  1
```

As we can see, from bitplane `A.high` to bitplane `A.low`, there is more and more alteration between 0 and 1. The bitplane `A.low` in this example features more random distribution of 0s and 1s, meaning less correlation among the pixels.

To achieve an effective compression, different compression methods should be applied to the bitplanes. For example, we could apply the Run-length method to *A.high*, and *A.middle*, and the Huffman coding to *A.low*.

Predictive encoding

The main idea is to predict the values of a pixel based on some previous pixels and to store the difference between the predicted and actual values in a residual image.

Example 10.5 *Suppose that the following matrix A[i,j] represents the pixel values of part of a larger grayscale image, where row indices* $i = 1, \cdots, 8$ *and column indices* $j = 1, \cdots, 8$*:*

```
2  4  2  3  1  1  1  1
1  3  2  3  5  1  1  1
3  4  5  5  5  5  5  5
6  4  5  8  7  9  4  5
3  2  7  3  2  7  9  4
3  3  4  3  4  4  2  2
1  2  1  2  3  3  3  3
1  1  1  2  2  3  3  3
```

Let us predict that each pixel is the same as the one to its left. So the residual matrix R[i,j]=A[i,j]-A[i,j-1] is as below, where $i = 1, \cdots, 8$, $j = 2, \cdots, 8$*:*

```
2  2 -2  1 -2  0  0  0
1  2 -1  1  2 -4  0  0
3  1  1  0  0  0  0  0
6 -2  1  3 -1  2 -5  1
3 -1  5 -4 -1  5  2 -5
3  0  1 -1  1  0 -2  0
1  1 -1  1  1  0  0  0
1  0  0  1  0  1  0  0
```

We can now apply any methods of prefix coding[1]*, to encode according to the frequency distribution of R.*

[1] *i.e. those text compression methods for generating a prefix code such as Huffman coding.*

The frequency distribution for this part of the residual image is then as below:

entry	occurrence	probability
-5	2	2/64
-4	2	2/64
-2	4	4/64
-1	6	6/64
0	21	21/64
1	16	16/64
2	6	6/64
3	4	4/64
5	2	2/64
6	1	1/64

The entropy of the distribution is $-\sum_i p_i log_2 p_i$ *bits. We can then use this entropy of partial image to code the entire image if the entropy is a good estimation for the whole image. Hopefully this would achieve a better compression in comparison to the original 8 bits/pixel (e.g. ASCII) coding.*

JPEG lossless coding

The predictive encoding is sometimes called *JPEG lossless coding* because the JPEG standard (codified as the ITU-T T.82 recommendation) specifies a similar predictive lossless algorithm.

The algorithm works with 7 possible predictors (i.e. the prediction schemes). The compression algorithm chooses a predictor which maximises the compression ratio. Given a pixel pattern (see below), the algorithm predicts the pixel '`x?`' in one of 8 ways:

T	S
Q	x?

1. No prediction[2]
2. $x = Q$
3. $x = S$
4. $x = T$
5. $x = Q + S - T$
6. $x = Q + (S - T)/2$
7. $x = S + (Q - T)/2$
8. $x = (Q + S)/2$

[2] *i.e. using the original pixel value matrix (A in Example 10.5), but this tends to be ineffective.*

The lossless JPEG[3] algorithm gives approximately 2:1 compression ratios on typical grayscale and colour images, which is well superior to GIF (although GIF may be competitive on icons or the like).

[3] *It is generally for lossy standard but does have the lossless version and It is an extensive and complicated standard for common use.*

Lossy compression

Lossy compression aims at achieving a good compression ratio, but the cost for it is the loss of some original information. Therefore, the measure of a lossy compression algorithm normally includes a measure of the quality of reconstructed images compared with the original ones.

Distortion measure

Although the best measure of the closeness or fidelity of a reconstructed image is to ask a person familiar with the work to actually look at the image and provide an opinion, this is not always practical because it is not useful in mathematical design approaches.

Here we introduce the more usual approach in which we try to measure the difference between the reconstructed image and the original one.

There are mathematical tools that measure the distortion in value of two variables, also called *difference distortion measure*. Considering an image to be a matrix of values; the measure of lossy compression algorithm normally uses a standard matrix to measure the difference between a reconstructed images and the original ones.

Let P_i be the pixels of reconstructed image and Q_i be the ones of the original, where $i = 1, \cdots, N$. We have the following commonly used measures:

- Squared error measure matrix (this is a measure of the difference):

$$D_i = (P_i - Q_i)^2$$

- Absolute difference measure matrix:

$$D_i = |P_i - Q_i|$$

- Mean squared error measure (MSE) matrix (this is an average measure):

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(P_i - Q_i)^2$$

- Signal-to-noise-ratio (SNR) matrix (this is the ratio of the average squared value of the source output and the MSE):

$$\text{SNR} = \frac{\sigma_x^2}{\sigma_d^2}$$

  This is often in logarithmic scale (dB):

$$\text{SNR} = 10\log_{10}\frac{\sigma_x^2}{\sigma_d^2}$$

- Peak-signal-to-noise-ratio (PSNR) matrix (this measures the error relative to the average squared value of the signal, again normally in logarithmic scale (dB)):

$$\text{SNR} = 20\log_{10}\frac{max_i|P_i|}{\sigma_d^2}$$

- Average of the absolute difference matrix:

$$D_i = \frac{1}{N}\sum_{i}^{N}|P_i - Q_i|$$

- Maximum value of the error magnitude matrix:

$$D_{i\infty} = \max_n D_i = |P_i - Q_i|$$

Progressive image compression

The main idea of progressive image compression is to gradually compress an image following a underlined order of priority. For example, compress the most important image information first, then compress the next most important information and append it to the compressed file, and so on.

This is an attractive choice when the compressed images are transmitted over a communication channel, and are decompressed and viewed in real time. The receiver would be able to view a development process of the image on the screen from a low to a high quality. The person can usually recognise most of the image features on completion of only 5-10% of the decompression.

The main advantages of progressive image compression are that:

1. The user can control the amount of loss by means of telling the encoder when to stop the encoding process
2. The viewer can stop the decoding process early since she or he can recognise the Image's feature at the early stage
3. as the compressed file has to be decompressed several times and displayed with different resolution, the decoder can, in each case, stop the decompression process once the device resolution has been reached.

There are several ways to implement the idea of progressive image compression:

1. Using so-called SNR progressive or quality progressive compression (i.e. encode spatial frequency data progressively).
2. Compress the gray image first and then add the colour. Such a method normally features slow encoding and fast decoding.
3. Encode the image in layers. Early layers are large low-resolution pixels followed by later smaller high-resolution pixels. The progressive compression done in this way is also called *pyramid coding* or *hierarchical coding*.

Example 10.6 *The following methods are often used in JPEG:*

- *Sequential coding (baseline encoding): this is a way to send data units following a left-to-right, top-to-bottom fashion.*
- *Progressive encoding: this transmits every nth line before filling the data in the middle.*
- *Hierarchical encoding: this is a way to compress the image at several different resolutions.*

Transforms

Transform is a standard mathematical tool used to solve complicated computation problems in many areas. The main idea is to change a quantity such as a number, vector or a function to another form in which some useful features may occur (e.g. the computation in the new form may be easier, or becomes more feasible). The result is then transformed back to the original form.

An image can be compressed if its *correlated* pixels are transformed to a new representation where the pixels become less correlated, i.e. are *decorrelated*. Compression is successful if the new values are smaller than the original ones on average. Lossy compression can be achieved by quantisation of the transformed values. The decoder normally reconstructs the original data from the compressed file applying the opposite transform.

Orthogonal transform

In this transform, matrix computation is applied. Let C and D be vectors and W be a $n \times n$ matrix and c_i, d_i and w_{ij} be the elements in C, D and W respectively. Further let $c_i = \sum_j w_{ij} d_j$, where c_i is a weighted sum of pixels d_j of an image and each pixel is multiplied by a weight w_{ij}, $i, j = 1, \cdots, 3$.

This can be written in matrix form:

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}$$

We can write $\mathbf{C} = \mathbf{W} \cdot \mathbf{D}$. Each row of $\mathbf{W}$ is called *basis vector*.

In practice, the weights should be independent of the values of d_is so the weights do not have to be included in the compressed file.

Example 10.7 *Given a weight matrix:*

$$W = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & 1 \end{pmatrix}$$

If we use the transfer vector

$$D = \begin{pmatrix} 4 \\ 6 \\ 5 \end{pmatrix}$$

Then we have

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 4 \\ 6 \\ 5 \end{pmatrix} = \begin{pmatrix} 15 \\ -7 \\ 3 \end{pmatrix}$$

Compare the energy of $(15, -7, 3)^T$ *to that of* $(4, 6, 5)^T$*. We have* $15^2 + (-7)^2 + 3^2 = 283$ *compared with* $4^2 + 6^2 + 5^2 = 77$*, which is nearly 4 times smaller.*

However, the original energy may be conserved if we adjust $\mathbf{W}$ *by multiplying a factor of 1/2. Now we have* $\mathbf{C} = (15/2, -7/2, 3/2)^T$*.*

When reconstructing the original $\mathbf{D}$*, we first quantify the transformed vector* $\mathbf{C}$ *from* $(7.5, -3.5, 1.5)^T$ *to integers* $(8, -4, 1)^T$ *and conduct an inverse transform:* $\mathbf{D} = \mathbf{W^{-1}} \cdot \mathbf{C}$

$$(1/2) \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & -1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 8 \\ -4 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 7 \end{pmatrix}$$

In practice, the $\mathbf{W}$ *is much bigger so the reconstruction result would be better. More sophisticated transform techniques would also produce better results.*

Wavelet transforms

Wavelet transform is one of the most popular forms of sound or video compressing signals. The two-dimensional Wavelet transform can be used for image compression. The main idea is similar to an extended version of the one discussed in the previous section. Wavelet transform can require a fair degree of mathematical sophistication to implement and has many tool kits available including an entire class of special processing chips and devices, known as digital signal processors. Due to time pressure, we only briefly sketch a general picture without details in this subject. Also, for convenience of discussion, we give the definition of one-dimensional transform only.

The Wavelet transform works by choosing a set of representative patterns and finding a subset of these patterns that add up to the signal. Let $f(t)$ be the signal and t is the measure of time. This might be a sound file that stores the intensity of the sound every millisecond.

Let $g_i(t)$ be another set of functions known as the *basis*. Suppose a function f can be represented as the weighted sum of some set of $g_i(t)$:

$$f(t) \approx \sum_{i}^{N} \alpha_i g_i(t)$$

The values of α_i are called the *coefficients*, which can be boolean, integer or real numbers. The compression process aims to find a good set of coefficients $(\alpha_1, \alpha_2, \cdots, \alpha_N)$ to represent signal f. The decompression process aims to reconstruct f from the set of coefficients. The compression and decompression algorithms share the same set of functions and the compressed file consists of the set of coefficients.

The main issues in a Wavelet transform are:

- finding a good set of basis functions for a particular class of signal
- finding an effective quantisation method for different frequencies.

At present, the effort and choices made for both of the issues are as much as an art rather than a science.

Due to time limits, we only list the most common approaches here:

- Discrete Cosine (Sine) transform (DCT)
- Fourier transform (FT)
- Walsh-Hadamard Transform (WHT)
- Haar Transform (HT)

Karhunen-Loeve Transform (KLT)

Interested readers may consult the books given at the beginning of the chapter for details.

JPEG (Still) Image Compression Standard

This is one of the most widely recognised standards in existence today. and provides a good example of how various techniques can be combined to produce fairly dramatic compression results. The baseline JPEG compression method has a wide variety of hardware and software implementations available for many applications.

JPEG has several lossy encoding modes, from so-called *baseline sequential mode* to *lossless encoding mode*.

The basic steps of lossy JPEG algorithm include processing 24 (or 32) bit colour images and offering a trade-off between compression ratio and the quality.

There are a lot of techniques and algorithms for image data compression. You may consult various sources in literature. Due to the time limit in the subject, we have only provided a sample of the techniques here.

Learning outcomes

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

- provide examples of lossless and lossy image compression techniques
- illustrate how to represent a grayscale image by several bi-level images
- describe the main ideas of predictive encoding
- be familiar with various distortion measurements
- explain the general approaches of various popular lossy image compression techniques such as progressive image compression and transforms.

Activities

1. Construct a Reflected Gray Code for decimal numbers $0, 1, \cdots, 15$.
2. Following the above question, provide an example to show how a grayscale image with 16 shades of gray can be separated into 4 bi-level images.
3. Suppose that the matrix $A[i, j]$ below represents the pixel values of part of a large grayscale image, where $i = 0..7$, $j = 0..7$, and $A[0, 0] = 4$. Let us predict that each pixel is the same as the one to its right.
   (a) Derive the residual matrix $R[i, j]$ of A
   (b) Derive the frequency table for R
   (c) Derive a variable length code, e.g. Huffman code for the partial image.
   (d) Discuss under what conditions the code can be used for the whole image.

```
4 8 4 8 1 1 1 1
1 2 4 6 5 1 1 1
8 4 5 5 5 5 5 5
2 4 8 5 7 9 5 5
2 4 6 7 7 7 9 9
2 2 2 3 4 9 7 3
3 3 6 6 6 7 7 7
7 7 7 7 6 7 8 8
```

4. Given the orthogonal matrix

$$W = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix}$$

   and a data vector

$$D = \begin{pmatrix} 4 \\ 6 \\ 5 \\ 2 \end{pmatrix},$$

   conduct a similar experiment to the approach in this chapter.
   (a) Fulfil a linear transform on D
   (b) Perform a reverse transform to get D'
   (c) Compare D' with the original D.
5. Implement the ideas above using a higher level programming language such as Java or C.

Laboratory

1. Design and implement a method in Java which takes a binary codeword and returns its RGC codeword.
2. Design and implement a method in Java which takes a matrix of integers and returns its residual matrix, assuming that each entry is the same as the one to its left.

Sample examination questions

1. Derive the RGC codeword for a decimal 5.

2. Explain, with an example, how to represent the part of a grayscale image below by 3 bitplanes:

```
000 001 011 011
001 001 001 010
011 001 010 000
```

Chapter 11

Video compression

Essential reading

Wayner, Peter *Compression Algorithms for Real Programmers.* (Morgan Kaufmann, 2000) [ISBN 0-12-788774-1]. Chapter 11.

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 5.

Video systems

These can be classified into two types: *analog video* and *digital video.*

Analog video

The essential function of an analog video is to display still pictures on a television receiver or a monitor screen one frame after another. The pictures are converted into an electronic signal by a raster scanning process. Conceptually, a screen is divided into horizontal lines. The picture we see is built up as a sequence of horizontal lines from top to bottom. Here the human persistence of vision plays an important part and makes the series of lines appear a (frame of) picture.

The following parameters determine the quality of pictures:

- Number of scanning lines
- Number of pixels per scan line
- Number of displayed frames per second (in fps): this is called the *frame rate*
- Scanning techniques.

For example, the screen has to be refreshed about 40 times a second, it flickers otherwise. Popular standards include Phase Alternating Line (PAL), Sequential Couleur Avec Memoire (SECAM) and (USA) National Television Systems Committee (NTSC). PAL is mainly used in most of Western Europe and in Australia, China and New Zealand. SECAM is used in France, in the former Soviet Union and in Eastern Europe. NTSC is used in North America, Japan, Taiwan, part of the Cribbean, and in South America.

A PAL or SECAM frame contains 625 lines, of which 576 are for pictures. An NTSC frame contains 525 lines, of which 480 are pictures. PAL or SECAM use a frame rate of 25 fps and NTSC 30 fps.

Digital video

Although the idea of digital video is simply to sample the analog signal and convert it into a digital form, the standard situation is inevitably quite complex due to the need to compromise the existing equipment in the past as well as new standards such as HDTV.

The sample standard is ITU-R BT.601 (more commonly known as CCIR

601), which defines sampling of digital video. It specifies a horizontal sampling picture format consisting of 720 luminance samples and two sets of 360 colour difference samples per line. The size of PAL screen frames is 768x576 and NTSC 640x480.

Besides the digital data from analog video sampling, digital video data to be compressed also includes the data streams produced by various video equipment such as digital video cameras and VTRs. Once the data is input into the computer there are many ways to process them.

Moving pictures

Video data can be considered as a *sequence* of still images changing with time. Use of human eyes' forgiving feature, video signals change the image 20 to 30 times per second. Video compression algorithms all tend to be lossy.

One minute of modern video may consist of 1,200 still images. This explains the need and the motivation for video compression.

Among different numbers of different video compression standards, two are the most important ones, namely *ITU-T H.261* and *MPEG. ITU-T H.261* is intended for use mainly for video conferencing and video-telephony, and *MPEG* is mainly for computer applications.

MPEG

- This is the standard designed by the Motion Picture Expert Group, including the updated versions MPEG-1, MPEG-2 MPEG-4, MPEG-7.
- Most popular standards include the ISO's MPEG-1 and the ITU's H.261.
- It is built upon the three basic common analog television standards: NTSC, PAL and SECAM
- There are two sizes of SIF (Source Input Format): SIF-525 (with NTSC video) and SIF-625 (for PAL video).
- The H.261 standard uses CIF (the Common Intermediate Format) and the QCIF (Quarter Common Intermediate Format).

As we can see, MPEG is greatly influenced by ITU-T standards and even includes some of its standards. This may be the reason that ITU-T H.261 is the more fully developed of the two.

Basic principles

Video is based on two characteristics of the video data:

- **spatial redundancy**[1] exists in each frame: this can be dealt with by the techniques for compressing still images.
- **temporal redundancy**[2] occurs because a video frame tends to be similar to its immediate neighbours.

[1] *Recall this means the correlation among neighbouring pixels in an image.*

[2] *Recall this means the similarity among neighbouring frames.*

Techniques for removing spatial redundancy are called *spatial compression* or *infra-frame* and techniques for removing temporal redundancy are called *temporal compression* or *inter-frame*. We focus on temporal compression here.

Temporal compression algorithms

In these algorithms, certain frames in a sequence are identified as *key frames*. These key frames are often specified to occur at regular intervals. The key

frames are either left uncompressed or are more likely to be spatially compressed. Each of the frames between the key frames is replaced by a so-called *difference frame* which records the differences between the original frames and the most recent key frame. Alternatively, the differences between the original frames and the preceding frame can also be stored in the difference frame depending on the sophistication of the decompressor.

Example 11.1 *Figure 11.1 shows a sequence of frames to be displayed during a time duration* $0, 1, \cdots, 11$.

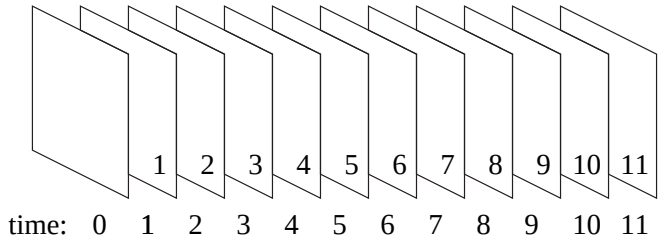


Figure 11.1: A sequence of original frames

Follows the MPEG's regulation, the frames marked 'I' (as shown in Figure 11.2) are encoded as the key frames. These are called *I-pictures*, meaning *intra* and having been compressed purely by spatial compression. Difference frames compared with previous frames are called *P-pictures* or *predictive pictures*. The frames marked *B* are so-called *B-pictures* which are predicted from later frames.

Example 11.2 *The encoded frames are as Figure 11.2.*

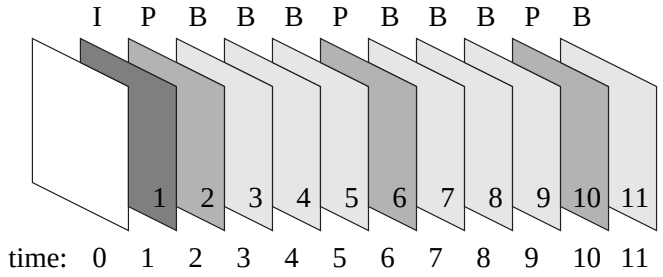


Figure 11.2: Encoded as I-,P- and B pictures

The decoding process now actually becomes a question of dealing with a sequence of compressed frames with I,P,B types of pictures.

I-pictures are decoded independently. P-pictures are decoded using the preceding I or P frames. B-pictures are decoded according to both preceding and following I- or P-pictures. The encoded frames can then be decoded as IBBBPBBBP and are displayed in that order (see Figure 11.4).

Example 11.3 *Note that the order of the encoded and displayed frames in Figure 11.3 are altered slightly from those in Figure 11.2.*

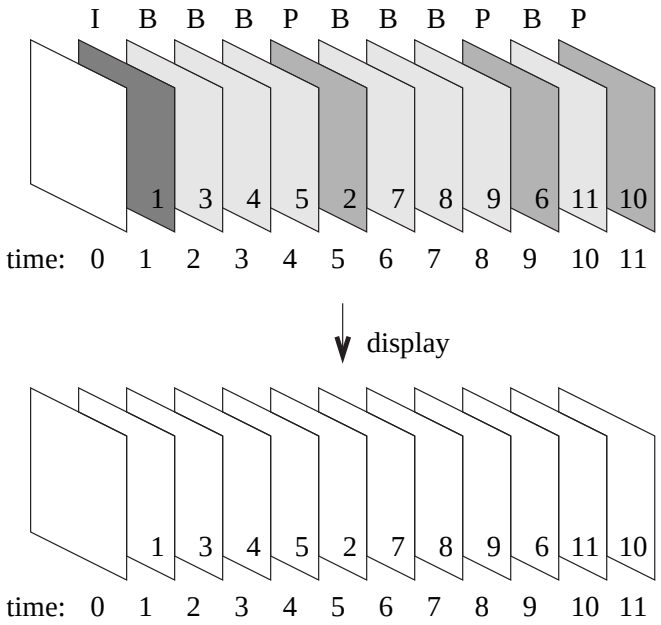


Figure 11.3: Decoded and displayed from I-,P- and B pictures

Group of pictures (GOP)

Now a video clip can be encoded as a sequence of I-,P- and B-pictures. GOP is a repeating sequence beginning by an I-picture and is used by encoders.

Example 11.4 *Figure 11.4 shows a GOP sequence* IBBPBB *containing 2 groups.*

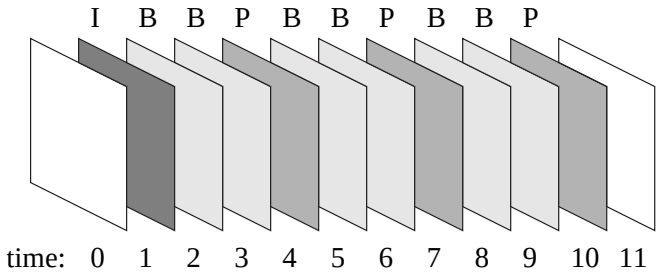


Figure 11.4:

Other common sequences in use are IBBPBBPBB with 3 groups and IBBPBBPBBPBB with 4 groups.

Motion estimation

The main idea is to represent each block of pixels by a vector showing the amount of block that has moved between images. The approach is most successful for compressing a background of images where the fixed scenery can be represented by a fixed vector of (0,0).

It uses the following methods:

- Forward prediction - computing the difference only from a previous frame that was already decoded.

- Backward prediction - computing the difference only from the next frame that is about to be decoded.
- Bidirectional prediction - the pixels' information in a frame depends on both the last frame just decoded and the frame about to follow.

The following factors can be used to estimate motion:

- Basic difference.
- L_1 norm - mean absolute error

$$||A - B||_1 = \sum_i \sum_j |A_{i,j} - B_{i,j}|.$$

- L_2 norm - mean squared error

$$||A - B||_2 = \sum_i \sum_j (A_{i,j} - B_{i,j})^2.$$

Work in different video formats

To allow systems to use different video formats, a process as follows is needed:

- Standardise the Format.
- Break the data into 8×8 blocks of Pixels.
- Compare to other Blocks.
- Choose important coefficients.
- Scale the coefficients for quantisation.
- Pack the coefficients.

Learning outcomes

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

- identify and list the parameters that determine the quality of pictures
- explain the popular standards for analog and digital videos
- describe, with an example, how temporal compression algorithms work in principle
- explain the common measures for motion estimation.

Activities

1. Explain briefly the concepts of spatial redundancy and temporal redundancy.
2. Implement the basic *temporal compression algorithm* introduced in the chapter.

Sample examination questions

1. Explain the two characters of video data.
2. Illustrate how a temporal compression algorithm works.

Chapter 12

Audio compression

Essential reading

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

Further reading

Salomon, David *A Guide to Data Compression Methods.* (Springer, 2001) [ISBN 0-387-95260-8]. Chapter 6.

Sound

Sound is essentially a sensation of the human audio system (i.e. it is detected by the ears and interpreted by the brain in a certain way). It is similar to colour in a way, a mixture of physical and psychological factors and it is therefore not easy to model accurately.

Nevertheless, we know that sound is a physical disturbance in a medium and it is propagated in the medium as a pressure wave by the movement of atoms or molecules. So sound can be described as a function (as for electronic signals), $f(t)$ measuring the pressure of medium at a time t. The easy function for this purpose is a sine wave $p(t) = \sin(t) = \sin(2\pi ft)$. The f is the frequency describing the number of cycles per second and it is measured in Hertz (Hz). If T is the period, the sine wave can be written as $p(t) = p(t+T)$ and $\sin(t) = \sin(\frac{2\pi}{T}t)$ since $f = \frac{1}{T}$.

Any periodic signal $s(t)$ with period T can be represented by the sum of sine or cosine waves:

$$s(t) = \frac{a_0}{2} + \sum_{i=1}^{\infty} a_i \cos(i\frac{2\pi}{T}t) + \sum_{i=1}^{\infty} a_i \sin(i\frac{2\pi}{T}t)$$

Example 12.1 *We know that most orchestras tune the A above middle C note to 440Hz. The sound pressure function for that note can be written therefore as* $p(t) = \sin(880\pi t)$.

We also know that the human ear is normally able to detect frequencies in the range between 20Hz and 20kHz. The upper audible limit in terms of frequency tends to decrease as age increases. Children may hear sounds with frequency as high as 20kHz but few adults can.

It is interesting to notice that the timbre of different instruments is created by the transient behaviour of notes. In fact, if the attack portion is removed from recordings of an oboe, violin, and soprano playing or singing the same note, the steady portions are indistinguishable.

Like any other wave functions, we can plot any sound by plotting its amplitude (pressure on medium) against time. The shape of the curve can demonstrate clearly the different properties of a specific sound.

Alternatively, we can use a *frequency spectrum diagram* to represent the changes of frequencies over a period of time. In a frequency spectrum diagram, we plot the amplitude against the frequency.

Example 12.2 *Figure 12.1 is a frequency spectrum diagram for* $s(t) = 0.1\sin(2\pi t) + 0.3\sin(8\pi t) + 0.25\sin(20\pi t)$.

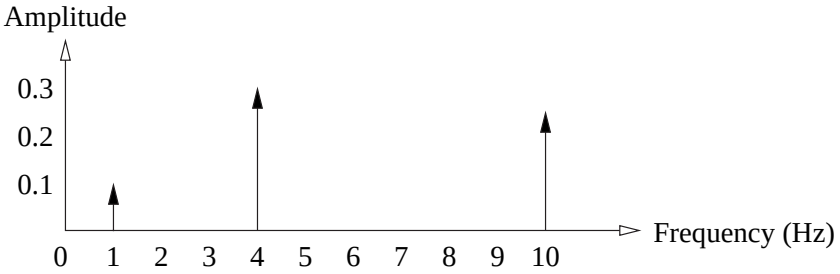


Figure 12.1: Frequency spectrum diagram

Digital sound data

Digitised audio data are the digital form of sounds. They are created by a process of *sampling* followed by *quantisation* on the analog sound wave.

Sampling

Sampling is a way of taking certain values at n discrete time $t_1, t_2, \cdots, t_n$. Samples taken at a time series with an equal time interval is called *sampling at a fixed rate*. The sampling frequency rate is simply $1/t$.

Otherwise, sampling taken with various different time intervals are called *sampling at a variable rate*. In this case the sampling frequency can be measured by the number of samples taken per second.

Example 12.3 *The sampling rate used for audio CDs is 44.1 kHz, 48 kHz is used for DAT (digital audio tape). 22.05 kHz is commonly used for audio signal for delivery over the Internet, and 11.025 kHz may be used for speech.*

Nyquist frequency

It is not difficult to understand that the choice of sample frequency rate can directly affect the quality of the digital sound data and subsequently the quality of compression. According to Nyquist theory, if a continuous wave form is band-limited to a frequency f then it must be sampled at a frequency of *at least* $2f$ in order to be able to reproduce the original waveform exactly.

Digitisation

Usually the values achieved from sampling are real numbers. Specifically, we need to convert the values into integers within the range [0,k-1] for some suitable integer k.

For sampling or digitisation, we may use the techniques introduced in previous chapters, or other more efficient techniques.

Audio data

If we plot the amplitude over a period of time for both the sound generated by speech and by a piece of music, we would find that the shape of the two waveforms very different. This suggests that we should deal with them separately.

Two types of sounds with very different characteristics are our main consideration here: *speech* and *music*. They are the most commonly used types of sound in multimedia productions today.

Representations specific to music and speech have been developed to effectively suit their unique characteristics. Speeches are represented in models of sounds, but music is represented as instructions for playing virtual instruments.

This leads to two compression areas, namely *voice compression* and *music compression*. Voice compression aims at removing the silence and music compression at finding an efficient way to reconstruct music at events.[1] In general, audio compression approaches are lossy.

[1] *The term 'event' is used to mean any form of music, not merely live social events*

Speech compression

This is also called *voice compression*.

We only briefly introduce a few commonly used compression methods here.

Pulse code modulation (ADPCM)

This technique is well known in other research areas such as telecommunication and networking. ADPCM stands for Adaptive Differential Pulse Code Modulation. It was formalised as a standard for speech compression in 1984 by ITU-T.

The standard specifies compression of 8 bit sound (typical representation for speech) sampled at 8 kHz to achieve a compression ratio of 2:1.

Speech coders

Here two major types of audio data are considered:

- telephone speech
- wideband speech

The goal is to achieve a compression ratio of 2:1 or better for both types of data. More specifically, the aim is to compress telephone speech to the compressed bit rate of less than 32 kbps, and to 64 kbps for wideband speech.

The following constraints make it possible to fulfil some of the tasks:

- The human ear can only hear certain sounds in normal speech.
- The sounds produced in normal human speech are limited.
- Non-speech signals on telephone lines are noises.

Two coders are used in practice, namely *waveform coders* and *vocoders* (voice coders). They both apply detailed models of the human voice tract to identify certain types of patterns in human speech.

Predictive approaches

This is similar to our approaches for video compression. The idea is to try to predict the next sample based on the previous sample and code the differences between the predicted and the actual sample values.

A basic version of the implementation would simply apply entropy coding, such as Huffman or arithmetic, to the differences between successive samples.

This approach is used in so-called VOCPACK algorithm for compressing 8 bit audio files (.wav).

Example 12.4 *Given a series of sample data,*
(27,29,28,28,26,27,28,28,26,25,27).
We can write the difference (2,-1,0,-2,1,1,0,-2,-1,2).
Now the entropy of the distribution of differences is $\log_2 5 = 2.32$ *bits. This would give a compression ratio of 8:2.23 (i.e. 3.45:1 if this can apply to the entire file).*

Of course, sampling is also a critical step to achieve good compression and it is always worth considering.

Music compression

Compression algorithms for music data are less well developed than that for other types of data.

The following facts contribute to this situation:

- There are various formats for digital sounds on computer.
- The development of digital audio has taken place in recording and broadcast industries.
- There are currently three major sound file formats: AIFF of MacOS, WAV or WAVE for Windows and AU (NeXT/Sun audio file format). Each of the three have evolved over the years and now provide similar capabilities in terms of the sampling rates and sizes and CD and DAT standard value storage, while MP3 in its own file format cannot accommodate sound compression by any other method.

We usually use a high bit rate and often attempt to capture a wider range of frequencies (20 to 20k Hz). It is usually difficult to decide what is a good compression system because this depends on each person's hearing ability. For example, to most people, lower frequencies add bass resonance and the result would sound more like the human voice. This, however, may not be the case at all for others.

Streaming audio

The idea of streaming audio is to deliver sound over a network and play it as it arrives without having to be stored on the user's computer first. This approach is more successful in general for sound than it is for video due to the lower requirement for bandwidth.

The available software includes:

- Real Networks' RealAudio (companion to RealVideo)

- Streaming QuickTime
- 'lo-fi' MP3

These are used already for broadcasting live concerts, for the Internet equivalent of radio stations, and for providing a way of playing music on the Internet.

MIDI

The main idea of MIDI compression is, instead of sending a sound file, to send a set of instructions for producing the sound. Of course, we need to make sufficiently good assumptions about the abilities of the receiver to make sure the receiver can play back the sound following the instruction, otherwise the idea cannot work in practice.

MIDI stands for Musical Instruments Digital Interface. It was originally a standard protocol for communicating between electronic instruments. It allowed instruments to be controlled automatically by sound devices that could be programmed to send out MIDI instructions.

MIDI can control many sorts of instruments, such as synthesisers and samplers, to produce various sounds of real music instruments.

Examples of available software include:

- QuickTime
- Cakewarlk Metro
- Cubse

QuickTime incorporates MIDI-like functionality. It has a set of instrument samples and can incorporate a superset of the features of MIDI. It can also read standard MIDI files, so any computer with it installed can play MIDI music without any extra requirement.

Note: One should realise that sound tends to work together with pictures or animation. In future most audio work has to take any potential synchronisation into consideration. For example, sound divided into video frames by some time code would be a useful function for film editing.

Learning outcomes

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

- explain how sound can be represented by a periodic function
- illustrate sound by frequency spectrum diagram as well as the normal plot for periodic functions
- describe the concept or principle of terms in audio data compression, such as ADPCM
- outline the distinction between voice compression and music compression in terms of the issues concerned in audio compression
- describe the main ideas of MIDI compression.

Activities

1. Given that singing has characteristics of both speech and music, which compression algorithms would you expect to be most successful on songs?

2. Given a frequency spectrum diagram as in Figure 12.2, write the signal $s(t)$ in its analog form, i.e. represented by the sum of sine and cosine waves.

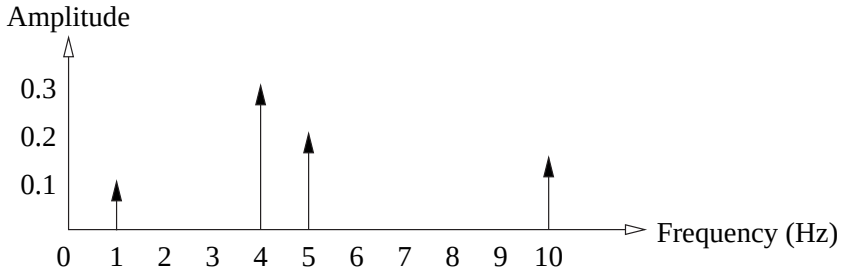


Figure 12.2:

3. Explain what MIDI stands for. Write a short essay of about 500-1000 words to introduce any application software which uses MIDI in one way or another.

Sample examination questions

1. Explain briefly the following terms:

   (a) Sampling

   (b) ADPCM

   (c) MIDI

2. Explain how a frequency spectrum diagram can be used to represent function $f(t) = 0.5 + sin(880\pi t) + sin(1760\pi t)$.

Chapter 13

Revision

Revision materials

You should make a good use of the following:

- lecture notes
- subject guide
- lab exercises
- exercises
- courseworks
- text book(s)
- any other handouts

Other references

You are advised to review mainly the revision materials listed above.

If time permits, you may also like to look up some concepts or different versions of algorithms introduced in lectures from various books. However, this is *not essential* for the examination purpose.

Examination

The duration of the examination will be *two hours and 15 minutes*.

There will be *five* questions and you must answer *three* out of the five questions[1]. There will be 75 marks available on the examination paper, but your marks will be converted to a right percentage later, of course.

[1] *The "three-out-of-five" pattern starts from 2004*

There will be a number of sub-questions in each question.

Questions in examination

The questions in the examination can be classified into the following three types:

- **Bookwork** The answers to these questions can be found in the *lecture notes* listed above.
- **Similar Question** The questions are similar to examples in the *revision materials* listed above. These are usually questions that require examples to be provided in addition to the definitions and explaining some concepts.
- **Unseen Question** You may have not seen these types of questions before the exam, but you should be able to answer these questions using the knowledge and experience gained from the course. The questions require some deeper understanding of the knowledge from the students.

The questions can also be classified according to the level of requirements to the student. For example,

- **Definition question** the answer to these questions require merely basic knowledge or definition in *lecture notes*. Questions are likely to

begin with 'What is xxx .. ?', 'Explain what xxx is ..'.

- **Conceptual question** This type of questions require from the student a good understanding of some important concepts. The questions seek mainly some concepts to be reviewed over what you have learnt. The questions are likely to start with 'Discuss xxx .. ?', 'Explain why xxx ..'. The answers to these questions should include not only the definition, theorem etc. learnt from the course, but also supporting examples to demonstrate good understanding of issues related to the concepts.
- **Do-it-yourself question** These questions require deeper understanding than that for Conceptual questions. The answers to the questions require not only the knowledge gained from the course but also certain degree of problem-solving skills involved.
  The questions normally consist of instructions to complete certain tasks. For example, 'Write a piece of Java code to ...', 'Draw a diagram for ...', 'Traverse the graph ...', or 'Analyse xxx ...' etc.

Read questions

Every year students are advised to read the questions on the examination paper *carefully*. You should make sure that you fully understand what is required and what subsections are involved in a question. You are encouraged to take notes, if necessary, while attempting the questions. Above all, you should be completely familiar with the course materials. To achieve a good grade, you need to have prepared well for the examination and to be able to solve problems by applying the knowledge gained from your studies of the course.

Recommendation

The following activities are recommended in your revision:

1. Write the definitions and new concepts learnt from the course in your own words.
2. Find a good example to support or help explaining each definition or concept discussed in the class.
3. Review the examples given in the *Revision materials*. Try to solve some problems using the skills learnt from the course.

Important topics

These will be covered in the examination.

1. Introduction
   - Entropy
   - Prefix codes
   - Optimal codes
   - Expected length, i.e. average length, of a code
   - Unique decodability
2. Minimum redundancy coding
   - Run-length algorithms
   - Shannon-Fano coding
   - Huffman coding
   - Adaptive Huffman coding

3. Arithmetic coding
    - Algorithm
    - Probability trees
4. Dictionary based compression
    - LZW compression
    - Sliding window (LZ77) compression (knowledge)
5. Image lossless compression
    - Image data
    - Colour systems
6. Concepts and principles of Lossy compression
    - Predictive coding
7. Speech compression (knowledge)

Good luck!

I hope you find this subject guide stimulating and useful, and you have enjoyed the studies as much as I do. Please feel free to let me know (i.pu@gold.ac.uk) how you feel about the whole experience of studying this module, the good, the bad or both.

Finally, I wish you all the best in the exam!

Appendix A

Sample examination paper I

Data Compression

Duration: 2 hours and 15 minutes

You must answer **THREE** *questions only. The full mark for each question is 25. The marks for each subquestion are displayed in brackets (e.g. [3]).*

Electronic calculators may be used. The make and model should be specified on the script and the calculator must not be programmed prior to the examination.

Question A.1

1. *Determine whether the following codes for the alphabet* $\{a, b, c, d\}$ *are uniquely decodable. Give your reasons for each case.* *[8]*
   (a) $\{1, 011, 000, 010\}$
   (b) $\{1, 10, 101, 0101\}$
   (c) $\{0, 01, 011, 0111\}$
   (d) $\{0, 001, 10, 011\}$
2. *Consider a text model for a black-white source file. Under what probability distribution condition does a static Huffman encoding algorithm achieve the best performance? Under what condition on the source does the algorithm perform the worst? Give your reasons.* *[7]*
3. *Outline a simple version of the adaptive Huffman encoding algorithm.* *[5]*
4. *Illustrate how adaptive Huffman encoding works on a source of text* CAAABB. *[5]*
   Hint:
   (a) *You may demonstrate step by step the progress of running the Adaptive Huffman encoding algorithm in terms of the* input, output, alphabet *and the* tree structure.
   (b) *Add sufficient comments in your algorithm if it is different from the one discussed in lecture.*

Question A.2

1. *Consider part of a grayscale image with 16 shades of gray that is represented by the array* A *below:*

   ```
   A:  0011 1000 1000 0010
       1100 1000 1100 0110
       1000 1100 1001 1001
   ```

   Demonstrate how the image can be represented by several bitplanes (bi-level images) *[4]*

2. *Explain, with the aid of an example, why a Huffman code is in general not optimal unless the probability of every symbol in the source alphabet is a negative power of 2.* *[5]*

3. *One way to improve the efficiency of Huffman coding is to maintain two sorted lists during the encoding process. Using this approach, derive a canonical minimum variance Huffman code for the alphabet {A,B,C,D,E,F} with the probabilities (in %) 34,25,13,12,9,7 respectively.* *[8]*

4. *Explain, with the aid of an example, each of the following terms:* *[8]*

   (a) *fixed-to-variable model*

   (b) *gray-scale image*

Question A.3

1. *Explain what is used to represent the so-called* colour depth *in a common RGB colour model. What is the value of the colour depth in a representation where 8 bits are assigned to every pixel?* *[4]*

2. *Comment on the truth of the following statement describing the absolute limit on lossless compression.* *[4]*

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

3. *Explain briefly what is meant by Canonical and Minimum-variance Huffman coding, and why it is possible to derive two different Huffman codes for the same probability distribution of an alphabet.* *[4]*

4. *Describe briefly, with the aid of an example, how Shannon-Fano encoding differs from static Huffman encoding.* *[5]*

   Hint: You may give an example by deriving a code for $\{A,B,C,D\}$ *with probability distribution 0.5, 0.3, 0.1, 0.1 using the two algorithms, and show the difference.*

5. *A binary tree (0-1 tree) can be used to represent a code containing a few codewords of variable length. Consider each of the four codes ((i)-(iv) below) for alphabet* $\{A,B,C,D\}$ *and draw the binary tree for each code.*

   (a) $\{000, 001, 110, 111\}$

   (b) $\{110,111,0,1\}$

   (c) $\{0000,0001,1,001\}$

   (d) $\{0001,0000,0001,1\}$

   For each tree drawn, comment on whether the code being represented by the binary tree is a prefix code and give your reasons. *[8]*

Question A.4

1. *Explain the meaning of the following terms:* *[8]*
   (a) *rate-distortion problem*
   (b) *entropy*
   (c) *prefix codes*
   (d) *motion prediction.*
2. *Given an alphabet of four symbols {A,B,C,D}, discuss the possibility of a uniquely decodable code in which the codeword for A has length 1, that for B has length 2 and for both C and D have length 3. Justify your conclusions.* *[4]*
3. *Derive the output of the HDC algorithm on the source sequence below. Explain the meaning of each control symbol that you use. Finally, describe briefly a data structure(s) and algorithm(s) that can be used in solving counting subproblems in the HDC algorithm.* *[5]*

   TTU␣␣␣␣␣␣␣␣␣K␣␣RR33333333333333␣␣PPPEE

   Hint: You may simply describe the data structure(s) and algorithm(s) that you have used in your Lab Exercise when implementing the HDC algorithm.
4. *Explain the concept of* bitmapped image*s and* vector graphic*s. What are the differences between vector and bitmapped graphics in terms of requirements to the computer storage capacity, and the size of the image files.* *[8]*

Question A.5

1. *It has been suggested that the unique decodability is not a problem for any fixed length code. Explain why this is so with the aid of an example.* *[4]*

2. *Describe the main idea of predictive encoding. Suppose the matrix below represents the pixel values (in decimal) of part of a grayscale image. Let the prediction be that each pixel is of the same value as the one to its left. Illustrate step by step how predictive encoding may be applied to the array.* *[6]*

   ```
   1 1 1 1
   5 1 1 1
   5 5 5 5
   7 9 4 5
   ```

3. *Demonstrate step by step how the Basic LZW encoding and decoding algorithms maintain the same version of a dictionary without ever transmitting it between the compression and the decompression end, using a small source string* `BBGBGH` *as an example.* *[15]*

   The LZW encoding and decoding algorithms are given below.

   (a) *Encoding*

   ```
   1. word='';
   2. while not end_of_file
         x=read_next_character;
         if word+x is in the dictionary
            word=word+x
         else
            output the dictionary index for word;
            add word+x to the dictionary;
            word=x;
   3. output the dictionary number for word;
   ```

   (b) *Decoding*

   ```
   1. read a token x from compressed file;
   2. look up dictionary for element at x;
      output element
      word=element;
   3. while not end_of_compressed_file do
         read x;
         look up dictionary for element at x;
         if there is no entry yet for index x
            then element=word+first_char_of_word;
         output element;
         add word+first_char_of_element to the dictionary;
         word=element;
   4. end
   ```

Appendix B

Exam solutions I

Please be aware that these solutions are given in brief note form for the purpose of this section. Candidates for the exam are usually expected to provide coherent answers in full form to gain full marks. Also, there are other correct solutions which are completely acceptable but which are not included here.

Duration: 2 hours and 15 minutes

You must answer **THREE** *questions only. The full mark for each question is 25. The marks for each subquestion are displayed in brackets (e.g. [3]).*

Electronic calculators may be used. The make and model should be specified on the script and the calculator must not be programmed prior to the examination.

Note:

B for bookwork: The answer to the question can be found in the lecture notes or study guide;

S for similar: The question is similar to an example in the lecture notes or study guide;

U for unseen: The students may have not seen the question but good students should be able to answer it using their knowledge and experience gained from the course.

.....

Question A.1

1. *(S)*

   (a) *Yes, because this is a prefix code.* *[1+1/8]*

   (b) *No. Because, for example, 01011010101 can be decoded either as DCD or as DBBC (or DBAD).* *[1+1/8]*

   (c) *Yes, because each codeword starts with a 0.* *[1+1/8]*

   (d) *No, because for example, 00100010 can be decoded as BAAAC or AACBA .* *[1+1/8]*

2. *(U)*

   Let the probability distribution of the source be $\{p_B, p_W\}$. *[1/7]*

   When $p_B = p_W = 1/2$, *the static Huffman algorithms work best.* *[1/7]*

   The reason is that the average length of the Huffman code is equal to the entropy of the source. The Huffman code is optimal in this case.

   The average length of the codewords is 1 and the entropy of the source is $1/2(-\log_2 1/2)1/2(-\log_2 1/2)+ = 1$. *[2/7]*

When $p_B = 0$, $p_W = 1$ (or $p_B = 1$, $p_W = 0$), the static Huffman algorithms work worst. *[1/7]*

The average length of the codewords is 1 and the entropy of the source is $1(-\log_2 1) = 0$. *[2/7]*

3. *(S) The following algorithm outlines the encoding process:* *[3/5]*

```
1. Initialise the Huffman tree T containing
      the only node DAG.
2. while (more characters remain) do
      s:= next_symbol_in_text();
      if (s has been seen before) then output h(s)
      else output h(DAG) followed by g(s)
      T:= update_tree(T);
   end
```

The function update_tree *[2/5]*

```
1. If s is not in S, then add s to S; weight_s:=1;
   else weight_s:=weight_s + 1;
2. Recompute the Huffman tree for the new set
      of weights and/or symbols.
```

4. Example: (Please note that a different but correct process is possible) [5]

```
For example, let g(s) be an ASCII code for symbol s and
h(s) be the Huffman code for symbol s (this is available
from the Huffman tree on each step).

(0) Initialise
S={DAG(0)}

Tree (with a single node):
          DAG(0)


(1)
Read-input:   C
Output    :   h(DAG) ASCII("C")
          :   i.e.  1 ASCII("C") [or 0 ASCII("C")]

S={C(1), DAG(0)}

Tree:       1
          /    \
        C(1)   DAG(0)

(2)
Read-input:   A
Output    :   h(DAG) ASCII("A")
          :   i.e.  1 ASCII("A")

S={C(1), A(1), DAG(0)}

Tree:       2
          /    \
         1      C(1)
       /   \
     A(1)  DAG(0)

(3)
Read-input:   A
Output    :   00

S={A(2), C(1), DAG(0)}
```

```
Tree:       3
          /     \
        A(2)     1
               /   \
             C(1)  DAG(0)

(4)
read-input:   A
Output    :   0

A={A(3), C(1), DAG(0)}

Tree:       4
          /     \
        A(3)     1
               /   \
            C(1)  DAG(0)

(5)
read-input:   B
Output    :   h(DAG) ASCII("B")

A={A(3), C(1), B(1), DAG(0)}

Tree:       5
          /     \
       A(3)     2
              /    \
             1     C(1)
           /   \
         B(1)  DAG(0)

(6)
read-input:   B
Output    :   100

A={A(3), B(2), C(1), DAG(0)}

Tree:       6
          /     \
         3      A(3)
       /   \
     B(2)   1
          /   \
        C(1)  DAG(0)
```

Question A.2

1. *(S)* A *can be separated into 4 bitplanes as follows:* *[1/4 each]*

```
0 1 1 0          [1]
1 1 1 0
1 1 1 1

0 0 0 0          [2]
1 0 1 1
0 1 0 0

1 0 0 1          [3]
0 0 0 1
0 0 0 0

1 0 0 0          [4]
0 0 0 0
0 0 1 1
```

2. *(UB)*

   A Huffman code is optimal only if the average (expected) length of the code is equal to the entropy of the source.

   Consider a code with n codewords. Let the lengths of codewords of the code are $l_1, l_2, .., l_n$. The average length of the codewords is:

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

   where p_i is the probability of the codeword i.

   The entropy of the code is:

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

   [2/5]

   In comparison of the two formula, we find there are n items in both $\sum$s, so $l = H$ iff $l_i = \log_2 \frac{1}{p_i} = -\log_2 p_i$, for $i = 1, 2, \ldots, n$.

   Therefore, $p_i = -2^{l_i}$, l_i is a non-negative integer.

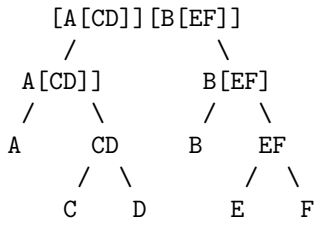
   As we can see, the probabilities have to be a negative power of 2. *[2/5]*

   An example (such as the case when $n = 4$). *[1/5]*

3. (S)

```
Linit: A  B  C  D  E  F
       34 25 13 12 9  7
Lcomb: empty

Linit: A  B  C  D
       34 25 13 12
Lcomb: EF
       16

Linit: A  B
       34 25
Lcomb: CD EF
       25 16

Linit: A
       34
Lcomb: B[EF] CD
       41    25

Linit: empty
Lcomb: A[CD]  B[EF]
       59     41

Linit: empty
Lcomb: [A[CD]][B[EF]]
       100
```

[2/8]



[2/8]

Assigning 0 to the left branch and 1 to the right. *[2/8]*

The canonical and minimum variance Huffman code is *[2/8]*

```
A 00
C 010
D 011
B 10
E 110
F 111
```

4. (UB)

(a) This is a model where a fixed number of symbols (usually one symbol) of the alphabet are assigned to a codeword and the code for the alphabet contains codewords of different length. In other words, the number of bits of the codeword for each group of fixed number

of symbols is a variable, i.e. the numbers of bits used for coding these units of fixed number of symbols are different. [2/8]

For example, Huffman coding belongs to this class. It assigns a longer codeword to the symbol that occurs less frequently in the source file, and a shorter codeword to the one that occurs more frequently. [2/8]

(b) Gray-scale image is a type of image with 8 bit colour depth. The images are captured by scanners using multiple intensity levels to record shadings between black and white. Each pixel is represented by 8 bits binary number which hold one colour value to provide 256 different shades of grey. [2/8]

Medical images and black and white photos are examples for gray scale images. An image consists of pixels and each pixel of the image is represented by a 8-bit shade value. For example, 00001000 can be used to represent a pixel with the shade of the value. [2/8]

Question A.3

1. *(U) The colour depth is represented by the number of bits used to hold a colour value in a RGB colour model.* *[2/4]*

   The value of the colour depth in this case: 8. *[2/4]*

2. *(S) Compressing a file can be viewed as mapping it to a different (hopefully a shorter) file. Compressing a file of n bytes (in size) by at least 1 byte means mapping it to a file of $n-1$ or fewer bytes. There are 256^n files of n bytes and 256^{n-1} of $n-1$ bytes in total. This means the mapping proportion is only $256^{n-1}/256^n = 1/256$ which is less than 1%.* *[4]*

3. *(U) If we follow the rules below during the encoding process, only one Huffman code will be derived and the code is called 'canonical'. Also, the codewords of the code will have the minimum variation in length.*

   i) A newly created element is always placed in the highest position possible in the sorted list.

   ii) When combining two items, the one higher up in the list is assigned a 0 and the lower down is a 1.

   Huffman coding that follows these rules is called Canonical and Minimum variance Huffman coding. *[2/4]*

   It would be possible to derive two different Huffman codes for the same probability distribution of an alphabet, because there can be items with equal probabilities at some stage on the sorted probability/frequency list, and this leads to two (or more) possible positions in which the symbols involved are placed in the tree, *[1/4]*

   and the roles of 0 and 1 are interchangeable depending on our 'rules'. *[1/4]*

4. *(U) The Shannon-Fano encoding and static Huffman coding algorithms both begin by creating one node for each character, but they build the tree differently. The Huffman method starts from the bottom (leaves) and builds the tree in a 'bottom-up' approach, while The Shannon-Fano method starts from the top (root) and builds the tree in a 'top-down' approach.* *[2/5]*

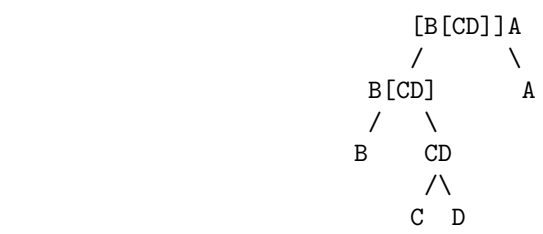
   Example: {A,B,C,D} with probability distribution 0.5, 0.3, 0.1, 0.1. *[3/5]*

   Huffman encoding:

   ```
   A   B   C   D
   0.5 0.3 0.1 0.1

   A   B   CD
   0.5 0.3 0.2
   ```

```
B[CD]  A
0.5    0.5

[B[CD]]A
1
```

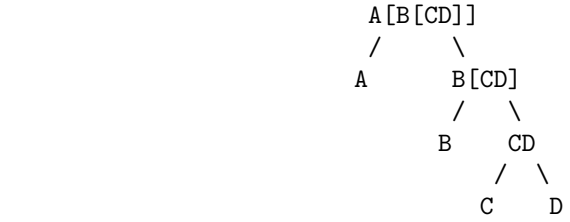
Huffman tree:



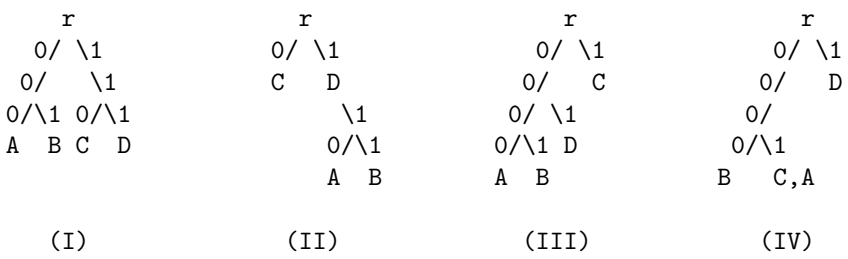
Shannon-Fano encoding

```
A    [  B   C   D   ]
0.5  [  0.3 0.1 0.1 ]

A    [  B   [C    D ]]
0.5  [  0.3 [0.1 0.1]]

A    [  B     [  C    D]]
0.5  [  0.3   [ 0.1  0.1]

A[B[CD]]
```

Shannon-Fano tree:



5. *(U) [(1/2+1/2+1)/8 each, for figure, prefix code and explanation]*



(I) is a prefix code, because all the symbols are at the leaves and no symbols share any node.

(II) is not a prefix code, because D is at an internal node. This means that the code for D is a prefix of A and B.

(III) is a prefix code, because all the symbols are at the leaves and no symbols share any node.

(IV) is not a prefix code, because A and C are represented by the same code (a prefix code should be uniquely decodable) which cannot be uniquely decodable. A and C share the same leaf although all symbols are leaves.

Question A.4

1. *(B)*

   (a) *rate-distortion problem*

   This is one of the two types of data compression problems (proposed by Davisson and Gray 1976). The compression problems of this type are to meet the requirement of achieving certain pre-specified fidelity with the best bit-rate possible, i.e. as few bits per second as possible. *[2/8]*

   (b) *entropy*

   The theoretical minimum average number of bits that are required to transmit a particular source stream is known as the entropy of the source and can be computed using Shannon's formula (in 1940s):

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

   [2/8]

   (c) *prefix codes*

   A prefix code is a code in which no codeword is a prefix to another codeword. This is when no sequence of bits which codes for one symbol is a prefix of the codeword for another symbol. *[2/8]*

   (d) *motion prediction*

   This is a class of basic compression techniques which are mainly for lossy image compression.

   The idea is to replace objects (e.g. a 8×8 block of pixels) in frames with references to the same object (at a slightly different position) in the previous frame. *[2/8]*

2. *(S)*

   Since $l_1 = 1, l_2 = 2,$ and $l_3 = l_4 = 3$. We have

   $$\sum_{i=1}^{4} = 1/2 + 1/2^2 + 1/2^3 + 1/2^3 = 1/2 + 1/4 + 1/8 + 1/8 = 1$$

   [2/4]

   As we see the Kraft-MacMillian's inequation is satisfied, so Yes, it is possible to find a uniquely decodable code for these lengths.

   [2/4]

3. *(S)*

   The output would be:

   $$n_3\texttt{TTU}r_9n_1\texttt{K}r_2n_2\texttt{RR}r_{14}\texttt{3}r_2r_3\texttt{P}n_2\texttt{EE}$$

   [2/5]

   Explanation of the control characters used: *[1/5]*

   n_3TTU*: 3 non-repeating characters TTU.*

   r_9*: 9 repeating blanks.*

   n_1K*: 1 non-repeating characters K.*

   r_2*: 2 repeating blanks.*

   n_2RR*: 2 non-repeating characters RR.*

r_{14}3: 14 repeating 3s.

r_2: 2 blanks.

r_3P: 3 repeating Ps.

n_2EE: 2 non-repeating characters EE.

Possible data structure(s) and algorithm(s):
Array, List, Tree, and a simple correct counting algorithm. *[2/5]*

4. *(BS) These are also called photographic images for 2-dimensional pictures. A bitmapped image is an array of pixel values. The data are the values of pixels. Many bitmapped images are created from digital devices such as scanners or digital cameras, or from programs such Painter which allows visual artists to paint images.* *[2/8]*

   The image in vector graphics is stored as a mathematical description of a collection of graphic components such as lines, curves and shapes etc..

   Vector graphics are normally more compact, scalable, resolution independent and easy to edit. They are very good for 3-D models on computer, usually built up using shapes or geometric objects that can easily be described mathematically. *[2/8]*

   The [requirements] to the computer system: a bitmapped image must record the value of every pixel, but vector description takes much less space in an economy way. *[2/8]*

   The [size] of bitmapped images depends on the size of image and the resolution but is independent from the complex of the image; while the size of vector graphics depends on the number of objects consist of the image, but is independent of any resolution. *[2/8]*

Question A.5

1. *(U) Because we can always use the fixed-length to decode a sequence of zeros and ones.* *[2/4]*

   For example, 01000100001001000100 can be easily divided into codewords if we know that each codeword is 4 bits long:

   0100,0100,0010,0100,0100. *[2/4]*

2. *The main idea of predictive encoding: is to predict the values of a pixel based on previous pixels and storing the difference between the predicted and actual values in a residual image.* *[2/6]*

   (a) *Derive the residual matrix:* *[2/6]*

   ```
   1  0  0  0
   5 -4  0  0
   5  0  0  0
   7  2 -5  1
   ```

   (b) *Use a prefix coding approach, such as Huffman coding (or any suitable compression technique):* *[2/6]*

   ```
   Frequency table:
   ----------------
   -5    1
   -4    1
   0     8
   1     2
   2     1
   5     2
   7     1
   ```

3. (U)

(a) Encoding: *[6/15]*

```
     word        x    (Output-token)  Dictionary
-------------------------------------------------------------------
                                       1   A  /* standard entries */
                                       2   B
                                       ...
                                       256 /
0    ‘’                                       /* initialisation */
.....................................................................
1                B
      B
.....................................................................
2                B     2              257 BB
      B
.....................................................................
3                G     2              258 BG
      G
.....................................................................
4                B     7              259 GB
      B
.....................................................................
5                G
      BG
.....................................................................
6                H     258            260 BGH
      H
.....................................................................
7                      8
     /* end of the algorithm */
-------------------------------------------------------------------
```

So the decoded string is 2 2 7 258 8.

(b) Decoding *[7/15]*

```
     word        x    (Output)          Dictionary
-------------------------------------------------------------------
                                       1   A /* standard entries */
                                       2   B
                                       .
                                       .
                                       256 /
.....................................................................
0                2    B
     B
.....................................................................
1                2    B                 257 BB
     B
.....................................................................
2                7    G                 258 BG
     G
.....................................................................
3                258  BG                259 GB
     BG
.....................................................................
4                8    H                 260 BGH
     H
     /* end of the algorithm */
-------------------------------------------------------------------
```

So the decoded string is BBGBGH.

(c) Conclusion: *[2/15]*

As we can see, the dictionary produced during encoding and the one during decoding are identical. Each dictionary is built independently without the need for transmission.

Appendix C

Sample examination paper II

Data Compression

Duration: 2 hours and 15 minutes

You must answer **FOUR** *questions only. The full mark for each question is 25. The marks for each subquestion are displayed in brackets (e.g. [3]).*

Electronic calculators may be used. The make and model should be specified on the script and the calculator must not be programmed prior to the examination.

Question C.1

1. *Explain the meaning of the following terms:* *[8]*
   (a) *rate-distortion problem*
   (b) *entropy*
   (c) *prefix codes*
   (d) *motion prediction.*
2. *Explain the concept of the following and provide one example for each case.* *[8]*
   (a) *fixed-to-variable model*
   (b) *gray-scale image*
3. *It has been suggested that unique decodability is not a problem for any fixed length codeword set. Explain why this is so with the aid of an example.* *[4]*
4. *Provide reasons, with the aid of an example, to support the following statement:* *[5]*

   "The Huffman code in general is not optimal unless all probabilities are negative powers of 2."

Question C.2

1. *Comment on the truth of the following statement in describing the absolute limits on lossless compression.* *[4]*

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

2. *Explain briefly what is meant by Canonical and Minimum-variance Huffman coding, and why it is possible to derive two different Huffman codes for the same probability distribution of an alphabet.* *[4]*

3. *Describe briefly, with the aid of an example, how Shannon-Fano encoding differs from static Huffman coding.* *[5]*

   Hint: You may give an example by deriving a code for {A,B,C,D} with probability distribution 0.5, 0.3, 0.1, 0.1 using the two algorithms, and show the difference.

4. *A binary tree (0-1 tree) can be used to represent a code containing a few codewords of variable length. Consider each of the four codes for characters A,B,C,D below and draw the binary trees for each code.*

   (a) {*000, 001, 110, 111*}

   (b) {*110,111,0,1*}

   (c) {*0000,0001,1,001*}

   (d) {*0001,0000,0001,1*}

   For each tree drawn, comment on whether the code being represented by the binary tree is a prefix code and give your reasons.

   [12]

Question C.3

1. *Given an alphabet of four symbols {A,B,C,D}, discuss the possibility of a uniquely decodable code in which the codeword for A has length 1, that for B has length 2 and for both C and D have length 3. Give your reasons.* *[4]*

2. *What would be the output if the HDC algorithm is applied to the sequence below? Explain the meaning of each control symbol that you use.* *[5]*

   TTU␣␣␣␣␣␣␣␣␣K␣␣RR33333333333333␣␣PPPEE

3. *One way to improve the efficiency of Huffman coding is to maintain two sorted lists during the encoding process. Using this approach, derive a canonical minimum variance Huffman code for the alphabet {A,B,C,D,E,F} with the probabilities (in %) 34,25,13,12,9,7 respectively.* *[8]*

4. *Explain the concept of* bitmapped image*s and* vector graphic*s. What are the differences between vector and bitmapped graphics in terms of requirements to the computer storage capacity, and the size of the image files.* *[8]*

Question C.4

1. *Determine whether the following codes for the alphabet {A, B, C, D} are uniquely decodable. Give your reasons for each case.* *[8]*

   (a) {*1, 10, 101, 0101*}

   (b) {*000, 001, 010, 111*}

   (c) {*0, 001, 10, 011*}

   (d) {*000, 010, 011, 1*}

2. *Consider a source with a binary alphabet {B, W}. Under what condition on the source does the static Huffman coding algorithm achieve the best performance? Under what condition on the source does the algorithm perform the worst? Give your reasons.* *[7]*

3. *Illustrate how adaptive Huffman encoding works.* *[6]*

   Hint: You may demonstrate step by step the progress of running an Adaptive Huffman encoding algorithm in terms of the input, output, alphabet *and the* tree structure. *Suppose the sequence of symbols to be encoded from the initial state is* CAAABB.

4. *Explain what is used to represent the so-called* colour depth *in a common RGB colour model. What is the value of the colour depth in a representation where 8 bits is assigned to every pixel?* *[4]*

Question C.5

1. *Huffman coding can perform badly when it is applied to fax images. For example, the canonical minimum-variance Huffman code is about 37% worse than its optimal when the probabilities of the white pixels* $Pr(White) = 0.2$ *and* $Pr(Black) = 0.8$*. Demonstrate step by step how this situation can be improved by blocking two symbols at a time from the source.* *[12]*

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

2. *Derive the reflected Gray code for decimal number 5.* *[3]*

3. *A binary sequence of length 5 (symbols) was encoded on the binary alphabet* $\{A,B\}$ *using Arithmetic coding. Suppose the probability* $Pr(A) = 0.2$*. If the encoded output is 0.24, derive the decoded output step by step.* *[10]*

   The decoding algorithm is given below.

```
1. (Initialise) L <- 0 and d <- 1
2. (Probabilities) For the next symbol,
     Let Pr[s1] be p1, the probability for s1
     and Pr[s2] be p2, the probability for s2
3. (Update) If x is a member of [L,L+d*p1)
     then output s1, leave L unchanged, and
          set d<-d*p1
     else if x is a member of [L+d*p1, L+d)
          then output s2, set L<- L+d*p1 and d<-d*p2
4. (Done?) If the_number_of_decoded_symbols
              < the_required_number_of_symbols
     then go to step 2.
```

Question C.6

1. *Consider part of a grayscale image with 16 shades of gray that is represented by the array* A *below:*

   ```
   0011 1000 1000 0010
   1100 1000 1100 0110
   1000 1100 1001 1001
   ```

   Demonstrate how the image can be represented by several bitplanes (bi-level images) *[4]*

2. *Describe the main idea of predictive encoding. Suppose the matrix below represents the pixel values (in decimal) of part of a grayscale image. Let the prediction be that each pixel is the same as the one to its left. Illustrate step by step how predictive encoding may be applied to it.* *[6]*

   ```
   1 1 1 1
   5 1 1 1
   5 5 5 5
   7 9 4 5
   ```

3. *Demonstrate step by step how the Basic LZW encoding and decoding algorithms maintain the same version of a dictionary without ever transmitting it in a separate file, using a small string* `AABABC` *as an example.* *[15]*

   The LZW encoding and decoding algorithms are given below.

   (a) *Encoding*

   ```
   word='';
   while not end_of_file
      x=read_next_character;
      if word+x is in the dictionary
         word=word+x
      else
         output the dictionary index for word;
         add word+x to the dictionary;
         word=x;
   output the dictionary number for word;
   ```

   (b) *Decoding*

   ```
   read a token x from compressed file;
   output element
     word=element;
   while not end_of_compressed_file do
        read x;
        look up dictionary for element at x;
        if there is no entry yet for index x
           then element=word+first_char_of_word;
        output element;
        add word+first_char_of_element to the dictionary;
        word=element;
   ```

Appendix D

Exam solutions II

Please be aware that these solutions are given in brief note form for the purpose of this section. Candidates for the exam are usually expected to provide coherent answers in full form to gain full marks. Also, there are other correct solutions which are completely acceptable but which are not included here.

Duration: 2 hours and 15 minutes

You must answer **FOUR** *questions. The full mark for each question is 25. The marks for each subquestion are displayed in brackets (e.g. [3]).*

Electronic calculators may be used. The make and model should be specified on the script and the calculator must not be programmed prior to the examination.

Question A.1

1. *(B)*
   (a) *rate-distortion problem*
   This is one of the two types of data compression problems (proposed by Davisson and Gray 1976). The compression problems of this type are to meet the requirement of achieving certain pre-specified fidelity with the best bit-rate possible, i.e. as few bits per second as possible. *[2/8]*
   (b) *entropy*
   The theoretical minimum average number of bits that are required to transmit a particular source stream is known as the entropy of the source and can be computed using Shannon's formula (in 1940s):
   $$H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$
   [2/8]
   (c) *prefix codes*
   A prefix code is a code in which no codeword is a prefix to another codeword. This is when no sequence of bits which codes for one symbol is a prefix of the codeword for another symbol. *[2/8]*
   (d) *motion prediction*
   This is a class of basic compression techniques which are mainly for lossy image compression.
   The idea is to replace objects (e.g. a 8×8 block of pixels) in frames with references to the same object (at a slightly different position) in the previous frame. *[2/8]*
2. *(UB)*
   (a) *This is a model where a fixed number of symbols (usually one symbol) of the alphabet are assigned to a codeword and the code for the alphabet contains codewords of different length. In other words,*

the number of bits of the codeword for each group of fixed number of symbols is a variable, i.e. the numbers of bits used for coding these units of fixed number of symbols are different. *[2/8]*

For example, Huffman coding belongs to this class. It assigns a longer codeword to the symbol that occurs less frequently in the source file, and a shorter codeword to the one that occurs more frequently. *[2/8]*

(b) Gray-scale image is a type of image with a 8 bit colour depth. The images are captured by scanners using multiple intensity levels to record shadings between black and white. Each pixel is represented by 8 bits binary number which hold one colour value to provide 256 different shades of grey. *[2/8]*

Medical images and black and white photos are examples for gray scale images. An image consists of pixels and each pixel of the image is represented by a 8-bit shade value. For example, 00001000 can be used to represent a pixel with the shade of the value. *[2/8]*

3. *(U) Because we can always use the fixed-length to decode a sequence of zeros and ones.* *[2/4]*

   For example, 01000100001001000100 can be easily divided into codewords if we know that each codeword is 4 bits long:

   0100,0100,0010,0100,0100. *[2/4]*

4. *(UB)*

   A Huffman code is optimal only if the average (expected) length of the code is equal to the entropy of the source.

   Consider a code with n codewords. Let the lengths of codewords of the code are $l_1, l_2, .., l_n$. The average length of the codewords is

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

   where p_i is the probability of the codeword i.

   The entropy of the code is

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

   . *[2/5]*

   In comparison of the two formula, we found there are n items in both $\sum s$), So $l = H$ iff $l_i = \log_2 \frac{1}{p_i} = -\log_2 p_i$, for $i = 1, 2, \ldots, n$.

   So $p_i = -2^{l_i}$, l_i is a non-negative integer.

   As we can see, the probabilities have to be a negative power of 2. *[2/5]*

   An example (such as the case when $n = 4$). *[1/5]*

Question A.2

1. *(S) Compressing a file can be viewed as mapping it to a different (hopefully a shorter) file. Compressing a file of n bytes (in size) by at least 1 byte means mapping it to a file of $n-1$ or fewer bytes. There are 256^n files of n bytes and 256^{n-1} of $n-1$ bytes in total. This means the mapping proportion is only $256^{n-1}/256^n = 1/256$ which is less than 1%.* *[4]*

2. *(U) If we follow the rules below during the encoding process, only one Huffman code will be derived and the code is called 'canonical'. Also, the codewords of the code will have the minimum variation in length.*

   i) A newly created element is always placed in the highest position possible in the sorted list.

   ii) When combining two items, the one higher up in the list is assigned a 0 and the lower down is a 1.

   Huffman coding that follows these rules is called Canonical and Minimum variance Huffman coding. *[2/4]*

   It would be possible to derive two different Huffman codes for the same probability distribution of an alphabet, because there can be items with equal probabilities at some stage on the sorted probability/frequency list, and this leads to two (or more) possible positions in which the symbols involved are placed in the tree, *[1/4]*

   and the roles of 0 and 1 are interchangeable depending on our 'rules'. *[1/4]*

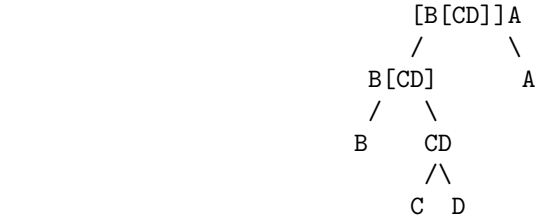
3. *(U) Shannon-Fano encoding and static Huffman coding algorithms both begin by creating one node for each character, but they build the tree differently. Huffman method starts from the bottom (leaves) and builds the tree in a 'bottom-up' approach, while Shannon-Fano method starts from the top (root) and builds the tree in a 'top-down' approach.* *[2/5]*

   Example: {A,B,C,D} with probability distribution 0.5, 0.3, 0.1, 0.1. *[3/5]*

   Huffman encoding:

```
A   B   C   D
0.5 0.3 0.1 0.1

A   B   CD
0.5 0.3 0.2

B[CD]  A
0.5    0.5

[B[CD]]A
```

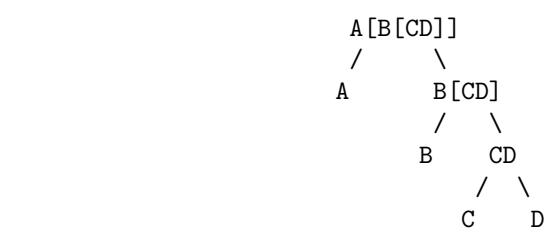
   Huffman tree:



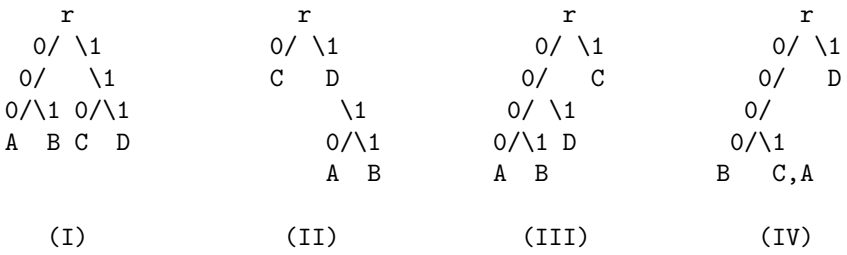
Shannon-Fano encoding

```
A    [  B   C   D   ]
0.5  [  0.3 0.1 0.1 ]

A    [  B   [C    D ]]
0.5  [  0.3 [0.1 0.1]]

A    [  B     [  C    D]]
0.5  [  0.3   [ 0.1  0.1]

A[B[CD]]
```

Shannon-Fano tree:



4. *(U) [(1+1+1)/12 each, for figure, prefix code and explanation]*



(I) a prefix code, because all the symbols are at the leaves and no symbols share any node.

(II) not a prefix code, because D is at an internal node. This means that the code for D is a prefix of A and B.

(III) a prefix code, because all the symbols are at the leaves and no symbols share any node.

(IV) not a prefix code, because A and C are represented by the same code (a prefix code should be uniquely decodable) which cannot be uniquely decodable. A and C share the same leaf although all symbols are leaves.

Question A.3

1. *(S)*

   Since $l_1 = 1, l_2 = 2,$ *and* $l_3 = l_4 = 3$. *We have*

   $$\sum_{i=1}^{4} = 1/2 + 1/2^2 + 1/2^3 + 1/2^3 = 1/2 + 1/4 + 1/8 + 1/8 = 1$$

   [2/4]

   As we see the Kraft-MacMillian's inequation is satisfied, so Yes, it is possible to find a uniquely decodable code for these lengths.

   [2/4]

2. *(S)*

   The output would be:

   $$n_3\texttt{TTU}r_9n_1\texttt{K}r_2n_2\texttt{RR}r_{14}\texttt{3}r_2r_3\texttt{P}n_2\texttt{EE}$$

   [2/5]

   Explanation of the control characters used: *[3/5]*

   n_3TTU*: 3 non-repeating characters TTU.*

   r_9*: 9 repeating blanks.*

   n_1K*: 1 non-repeating characters K.*

   r_2*: 2 repeating blanks.*

   n_2RR*: 2 non-repeating characters RR.*

   r_{14}3*: 14 repeating 3s.*

   r_2*: 2 blanks.*

   r_3P*: 3 repeating Ps.*

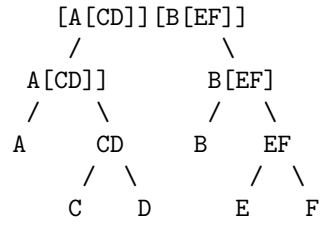
   n_2EE*: 2 non-repeating characters EE.*

3. *(S)*

```
Linit: A  B  C  D  E  F
       34 25 13 12 9  7
Lcomb: empty

Linit: A  B  C  D
       34 25 13 12
Lcomb: EF
       16

Linit: A  B
       34 25
Lcomb: CD EF
       25 16

Linit: A
       34
Lcomb: B[EF] CD
       41    25

Linit: empty
Lcomb: A[CD]  B[EF]
```

```
       59     41

Linit: empty
Lcomb: [A[CD]][B[EF]]
       100
```

[2/8]



[2/8]

Assigning 0 to the left branch and 1 to the right. *[2/8]*

The canonical and minimum variance Huffman code is *[2/8]*

```
A 00
C 010
D 011
B 10
E 110
F 111
```

4. *(BS) These are also called photographic images for 2-dimensional picture. A bitmapped image is an array of pixel values. The data are the values of pixels. Many bitmapped images are created from digital devices such as scanners or digital cameras, or from programs such Painter which allows visual artists to paint images.* *[2/8]*

   The image in vector graphics is store as a mathematical description of a collection of graphic components such as lines, curves and shapes etc..

   Vector graphics are normally more compact, scalable, resolution independent and easy to edit. They are very good for 3-D models on computer, usually built up using shapes or geometric objects that can easily be described mathematically. *[2/8]*

   The [requirements] to the computer system: a bitmapped image must record the value of every pixel, but vector description takes much less space in an economy way. *[2/8]*

   The [size] of bitmapped images depends on the size of image and the resolution but is independent from the complex of the image; while the size of vector graphics depends on the number of objects consist of the image, but is independent of any resolution. *[2/8]*

Question A.4

1. *(S)*

   (a) *No. Because, for example, 01011010101 can be decoded either as DCD or as DBBC (or DBAD).* *[1+1/8]*

   (b) *Yes, because all the codewords have the same length.* *[1+1/8]*

   (c) *No, because for example, 00100010 can be decoded as BAAAC or AACBA .* *[1+1/8]*

   (d) *Yes, because this is a prefix code.* *[1+1/8]*

2. *(U)*

   Let the probability distribution of the source be $\{p_B, p_W\}$. *[1/7]*

   When $p_B = p_W = 1/2$, *the static Huffman algorithms work the best. [1/7]*

   This is because that the average length of the Huffman code is equal to the entropy of source. The Huffman code is optimal.

   The average length of the codewords is 1 and the entropy of the source is $1/2(-\log_2 1/2)1/2(-\log_2 1/2)+ = 1$. *[2/7]*

   When $p_B = 0$, $p_W = 1$ *(or* $p_B = 1$, $p_W = 0$*), the static Huffman algorithms work the worst.* *[1/7]*

   The average length of the codewords is 1 and the entropy of the source is $1(-\log_2 1) = 0$. *[2/7]*

3. *(S) The following algorithm outlines the encoding process:* *[1/6]*

```
1. Initialise the Huffman tree T containing
      the only node DAG.
2. while (more characters remain) do
      s:= next_symbol_in_text();
      if (s has been seen before) then output h(s)
      else output h(DAG) followed by g(s)
      T:= update_tree(T);
   end
```

   The function `update_tree`

```
1. If s is not in S, then add s to S; weight_s:=1;
   else weight_s:=weight_s + 1;
2. Recompute the Huffman tree for the new set
      of weights and/or symbols.
```

Example: (Please note that a different but correct process is possible) *[5/6]*

```
For example, let g(s) be an ASCII code for symbol s and
h(s) be the huffman code for symbol s (this is available
from the Huffman tree on each step).

(0) Initialise
S={DAG(0)}

Tree (with a single node):
          DAG(0)


(1)
Read-input:   C
Output    :   h(DAG) ASCII("C")
          :   i.e.  1 ASCII("C") [or 0 ASCII("C")]

S={C(1), DAG(0)}

Tree:       1
          /     \
       C(1)   DAG(0)

(2)
Read-input:   A
Output    :   h(DAG) ASCII("A")
          :   i.e.  1 ASCII("A")

S={A(1), C(1), DAG(0)}

Tree:       2
          /     \
       A(1)      1
               /   \
            C(1)  DAG(0)

(3)
Read-input:   A
Output    :   00

S={A(2), C(1), DAG(0)}

Tree:       3
          /     \
       A(2)      1
               /   \
            C(1)  DAG(0)

(4)
read-input:   A
Output    :   0

A={A(3), C(1), DAG(0)}
```

```
Tree:       4
          /     \
        A(3)     1
                /   \
             C(1)  DAG(0)

(5)
read-input:   B
Output    :   h(DAG) ASCII("B")

A={A(3), C(1), B(1), DAG(0)}

Tree:       5
          /     \
       A(3)     2
                /    \
               1     C(1)
             /   \
          B(1)  DAG(0)

(6)
read-input:   B
Output    :   100

A={A(3), B(2), C(1), DAG(0)}

Tree:       6
          /     \
         3      A(3)
       /   \
     B(2)   1
          /   \
        C(1)  DAG(0)
```

4. *(U) The colour depth is represented by the number of bits used to hold a colour value in a RGB colour model.* *[2/4]*

   The value of the colour depth in this case: 8. *[2/4]*

Question A.5

1. *(U) This situation can be improved by the approaches below:*

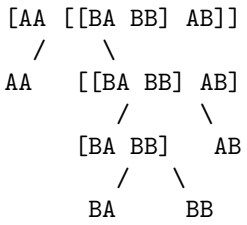
   (a) *By grouping 2 symbols at a time, the binary alphabet is extended to* $\{AA, AB, BA, BB\}$ *which consists of four elements.* *[2/12]*

   (b) *(Sub-total for this part* *[8/12])*

   The probabilities for the combined symbols are: *[2/12]*

```
AA 0.64
AB 0.16
BA 0.16
BB 0.04

AA   AB   BA   BB
0.64 0.16 0.16 0.04

AA   [BA BB]  AB
0.64 0.2      0.16

AA   [BA BB]  AB
0.64 0.2      0.16


AA   [[BA BB] AB]
0.64 0.36

[AA [[BA BB] AB]]
1
```

[2/12]



[2/12]

So the Huffman code:

```
AA  0
BB  11
BA  100
BB  101
```

[2/12]

(c) Now the average length is, for two symbols,

$$1 \times 0.64 + 2 \times 0.16 + 3 \times 0.16 + 3 \times 0.04 = 1.56 bits$$

This is 0.78 bits/symbol which is closer to the entropy 0.72 bit/symbol compared with before: average length was 1 bit/symbol.
[2/12]

2. *(UB) The binary code of 5 is 101.* *[1/3]*

   Shift 101 one bit to the right: 010 *[1/3]*

   The reflected Gray code for 5 is: 101 XOR 010 = 111 *[1/3]*

3. *(S) Let* s1=A, s2=B, $p_1 = 0.2$, $p_2 = 0.8$ *and* $x = 0.12$.

   The following table shows the values of main variables in each round. [8/10]

	L	d	d∗p1	d∗p2	[L, L+d∗p1)	[L+d∗p1, L+d)	Output
0	0	1					
1			0.2		[0, 0.2)	[0.2, 1)	B
	0.2	0.8		0.8			
2			0.16		[0.2, 0.36)		A
	0.2	0.16					
3			0.032	0.128	[0.2, 0.232)	[0.232, 0.36)	B
	0.232	0.128					
4			0.0256		[0.232, 0.2576)		A
	0.232	0.0256					
5			0.00512	0.02048	[0.232, 0.23712)	[0.23712, 0.2576)	B
	0.23712	0.0189696					

   So the decoded string is BABAB. *[2/10]*

Question A.6

1. A *can be separated into 4 bitplanes as following:* *[1/4 each]*

```
0 1 1 0          [1]
1 1 1 0
1 1 1 1

0 0 0 0          [2]
1 0 1 1
0 1 0 0

1 0 0 1          [3]
0 0 0 1
0 0 0 0

1 0 0 0          [4]
0 0 0 0
0 0 1 1
```

2. *Main idea of predictive encoding: is to predict the values of a pixel based on previous pixels and storing the difference between the predicted and actual values in a residual image.* *[2/6]*

   (a) *Derive the residual matrix:* *[2/6]*

```
1  0  0  0
5 -4  0  0
5  0  0  0
7  2 -5  1
```

   (b) *Use a prefix coding approach, such as Huffman coding (or any suitable compression technique):* *[2/6]*

```
Frequency table:
----------------
-5    1
-4    1
0     8
1     2
2     1
5     2
7     1
```

3. (U)

(a) Encoding: *[6/15]*

```
     word     x (Output-token)  Dictionary
-------------------------------------------------------
                                1   A  /* standard
                                       entries */
                                2   B
                                ...
                                256 /
0    ‘’                             /* initialisation */
.......................................................
1            A
      A
.......................................................
2            A     1           257 AA
      A
.......................................................
3            B     1           258 AB
      B
.......................................................
4            A     2           259 BA
      A
.......................................................
5            B
      AB
.......................................................
6            C     258         260 ABC
      C
.......................................................
7                    3
     /* end of the algorithm */
-------------------------------------------------------
```

So the decoded string is 1 1 2 258 3.

(b) Decoding *[7/15]*

```
     word        x    (Output)     Dictionary
-------------------------------------------------------
                                    1   A /* standard
                                        entries */
                                    2   B
                                    .
                                    .
                                    256 /
.......................................................
0                1     A
     A
.......................................................
1                1     A           257 AA
     A
.......................................................
2                2     B           258 AB
     B
.......................................................
3                258   AB          259 BA
     AB
.......................................................
4                3     C           260 ABC
     C
     /* end of the algorithm */
-------------------------------------------------------
```

So the decoded string is AABABC.

(c) Conclusion: *[2/15]*

As we can see, the dictionary produced during encoding and the one during decoding are identical. Each dictionary is built independently without the need for transmission.

Appendix E

Support reading list

This is a collection of textbooks in addition to those recommended at various points in the subject guide. It is for students' further interests on the subject and includes some historical backgrounds and useful web sites.

Text books

Drozdek, Adam *Elements of Data Compression.* (Brooks/Cole, USA, 2002) [ISBN 0-534-38448-X].

Gibson, Jerry D., Berger, Toby, Lookabaugh, Tom, Lindbergh, Dave, Baker, Richard L. *Digital Compression for Multimedia : Principles and Standards.* (San Francisco: Morgan Kaufmann Publishers, 1998) [ISBN 1-55860-369-7].

Nelson, Mark and Gailly, Jean-Loup *The Data Compression Book.* 2nd edition (New York: M&T Books, 1996) [ISBN 1-55851-434-1].

Ross, Sheldon, *A First Course in Probability* 3rd edition (New York: MacMillan, 1988) [ISBN 0-02-403850-4].

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

Web addresses

Data Compression
http://www.ics.uci.edu/~dan/pubs/DataCompression.html

Index

Notes

Comment form

We welcome any comments you may have on the materials which are sent to you as part of your study pack. Such feedback from students helps us in our effort to improve the materials produced for the International Programmes.
If you have any comments about this guide, either general or specific (including corrections, non-availability of Essential readings, etc.), please take the time to complete and return this form.

Title of this subject guide: .....

.....

Name .....

Address .....

.....

Email .....

Student number .....

For which qualification are you studying? .....

Comments

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Please continue on additional sheets if necessary.

Date: .....

Please send your completed form (or a photocopy of it) to:
Publishing Manager, Publications Office, University of London International Programmes,
Stewart House, 32 Russell Square, London WC1B 5DN, UK.