

# **Data Compression Techniques**

Report submitted in partial fulfillment of the requirement for the  
degree of

Bachelor of Technology.

in

**Computer Science & Engineering**

under the Supervision of

*Prof Dr. S.P. Ghrera*

By

*Vikram Thakur*

*111226*

to



Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

## **Certificate**

This is to certify that project report entitled “Data Compression Techniques”, submitted by Vikram Thakur(111226) in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science & Engineering to Jaypee University of Information Technology, Wagnaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

**Date:**

**Prof Dr. S.P. Ghreera**

**Designation: Professor**

## **Acknowledgement**

I am highly indebted to Jaypee University of Information Technology for their sources, guidance and constant supervision as well as for providing necessary information regarding the project & also for the support in completing the project.

I would like to express my gratitude towards my parents & Project Guide for their kind co-operation and encouragement which helped me in completion of this project.

I would like to express my special gratitude and thanks to industry persons for giving me such attention and time.

My thanks and appreciations also go to my colleagues in developing the project and people who have willingly helped me out with their abilities

Date:

Name : Vikram Thakur

## Table of Content

<b>S. No.</b>	<b>Topic</b>	<b>Page No.</b>
1.	Abbreviations	<i>vi</i>
2.	List of Figures	<i>vii</i>
3.	Abstract	<i>viii</i>
4.	Chapter 1 – Data Compression	1
4.1	Introduction to Compression	1
4.2	Motivation for Compression	2
5.	Chapter 2 – Concepts of Compression	4
5.1	Information theory	4
5.2	Data Compression Model	6
5.3	Semantic Dependent Methods	10
6.	Chapter 3 – Various Compression Algorithms	14
6.1	Shannon-Fano Coding	14
6.2	Static Huffman Coding	16
7.	Chapter 4 – Adaptive Huffman Coding	23
7.1	FGK Algorithm	24
7.2	Vitter Algorithm	27
7.3	Execution	30
8.	Chapter 5 – Graphs - Background	33

8.1	Introduction	33
8.2	Kinds of Graphs	35
8.3	Representing graphs in a computer	40
9.	Chapter 6 – Graph Compression	48
9.1	Introduction	48
9.2	Simple Weighted Graph Compression	51
9.3	Merge Algorithm	53
9.3	Compression Algorithm	54
10.	Conclusion and Future Work	57
11.	References	58

## Abbreviations and Symbols

1. **JPEG** – Joint Photographic Experts Group
2. **GIF** – Graphics Interchangeable Format
3. **HDTV** – High Definition Television
4. **MPEG-2** -Motion Pictures experts Group
5. **FFT** –Fast Fourier Transform
6. **UNIX** - Uniplexed Information and Computing System
7. **ASCII** - American Standard Code for Information Interchange
8. **EBCDIC** - Extended Binary Coded Decimal Interchange Code
9. **IBM** – International business Machines
10. **IMS** - Information Management System
11. **FORTRAN** -Formula Translation
12. **COBOL** –Common Business Oriented language
13. **PL/I** - Programming Language One
14. **FGK** – Faller, Gallager and Knuth
15. **V** - Vitter
16. **BSTW** -Bentley, Sleator, Tarjan and Wei

## List of Figures

<b>S. No.</b>	<b>Title</b>	<b>Page No.</b>
1.	Fig 1.1 – A block-block code	4
2.	Fig 1.2 – A variable code	4
3.	Fig 3.1 – A Shannon-Fano Code	15
4.	Fig 3.2 - Shannon-Fano <i>EXAMPLE</i>	15
5.	Fig 3.3 – Huffman process – The List	16
6.	Fig 3.3 Huffman Process – The Tree	17
7.	Fig 4.1(a) FGK Tree – After Processing <i>aa bb</i>	24
8.	Fig 4.1(b) – tree after encoding third <i>b</i>	25
9.	Fig 4.1( c) - Tree after update following first <i>c</i>	26
10.	Fig 4.2 – Tree for FGK for <i>EXAMPLE</i>	27
11.	Fig 4.5 – Algorithm Vitter processing <i>aabbb c</i>	30
12.	Fig 5.1 - Graph Illustration 1	34
13.	Fig 5.2 - Graph Illustration 2	35
14.	Fig 5.3 - Graph Illustration 3	36
15.	Fig 5.4 - Graph Illustration 4	37
16.	Fig 5.5 - Graph Illustration 5	38
17.	Fig 5.6 - Graph Illustration 6	39
18.	Fig 5.7 - Graph Illustration 7	42
19.	Fig 5.8 - Graph Illustration 8	44
20.	Fig 5.9 - Graph Illustration 9	45
21.	Fig 6.1 - Uncompressed Graph	50
22.	Fig 6.2 - Compressed Graph	50

## Abstract

Adaptive Huffman Coding is one of the many coding techniques used to compress data in a multitude of real world applications. GZip, 7Zip, Winrar are only a handful of utilities which make use of such data compression algorithms. Data Compression algorithms can be divided into 2 categories

1. Lossy Compression Algorithms
2. Lossless Compression Algorithms

Adaptive Huffman, which is derived from Huffman Coding, falls into the latter category and is extensively used in image formats such as JPEG (Joint Photographic Experts Group). There are a few shortcomings to the straight Huffman compression. First of all, you need to send the Huffman tree at the beginning of the compressed file, or the decompressor will not be able to decode it. This can cause some overhead.

Also, Huffman compression looks at the statistics of the whole file, so that if a part of the code uses a character more heavily, it will not adjust during that section. Not to mention the fact that sometimes the whole file is not available to get the counts from (such as in live information).

The solution to all of these problems is to use an Adaptive method.

It permits building the code as the symbols are being transmitted, having no initial knowledge of source distribution, that allows one-pass encoding and adaptation to changing conditions in data.

The benefit of one-pass procedure is that the source can be encoded in real time, though it becomes more sensitive to transmission errors, since just a single loss ruins the whole code.

Adaptive Huffman coding was first conceived independently by Faller and Gallager [Faller 1973; Gallager 1978]. Knuth contributed improvements to the original algorithm [Knuth 1985] and the resulting algorithm is referred to as algorithm FGK. A more recent version of adaptive Huffman coding is described by Vitter [Vitter 1987]. All of these



methods are defined-word schemes which determine the mapping from source messages to codewords based upon a running estimate of the source message probabilities.

The code is adaptive, changing so as to remain optimal for the current estimates. In this way, the adaptive Huffman codes respond to locality. In essence, the encoder is "learning" the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the encoder.

# CHAPTER 1

## DATA COMPRESSION

### 1. Introduction to Data Compression

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics in this domain, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs.

Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications. In this chapter we will use the generic term message for the objects we want to compress, which could be either files or messages. The task of compression consists of two components, an encoding algorithm that takes a message and generates a “compressed” representation (hopefully with fewer bits), and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation.

These two components are typically intricately tied together since they both have to understand the shared compressed representation. We distinguish between lossless algorithms, which can reconstruct the original message exactly from the compressed message, and lossy algorithms, which can only reconstruct an approximation of the original message.

Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters.

Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the

compression would be lossy since the text has changed, but the “meaning” and clarity of the message might be fully maintained, or even improved.

## **2. Motivation behind Compression**

Data compression has wide application in terms of information storage, including representation of the abstract data type string and file compression. Huffman coding is used for compression in several file archival systems [ARC 1986; PKARC 1987], as is Lempel-Ziv coding, one of the adaptive schemes. An adaptive Huffman coding technique is the basis for the compact command of the UNIX operating system, and the UNIX compress utility employs the Lempel-Ziv approach [UNIX 1984].

In the area of data transmission, Huffman coding has been passed over for years in favor of block-block codes, notably ASCII. The advantage of Huffman coding is in the average number of bits per character transmitted, which may be much smaller than the  $\log n$  bits per character (where  $n$  is the source alphabet size) of a block-block system.

The primary difficulty associated with variable-length codewords is that the rate at which bits are presented to the transmission channel will fluctuate, depending on the relative frequencies of the source messages. This requires buffering between the source and the channel. Advances in technology have both overcome this difficulty and contributed to the appeal of variable-length codes.

Current data networks allocate communication resources to sources on the basis of need and provide buffering as part of the system. These systems require significant amounts of protocol, and fixed-length codes are quite inefficient for applications such as packet headers. In addition, communication costs are beginning to dominate storage and processing costs, so that variable-length coding schemes which reduce communication costs are attractive even if they are more complex. For these reasons, one could expect to see even greater use of variable-length coding in the future.

It is interesting to note that the Huffman coding algorithm, originally developed for the efficient transmission of data, also has a wide variety of applications outside the sphere

of data compression. These include construction of optimal search trees, list merging and generating optimal evaluation trees in the compilation of expressions.

Additional applications involve search for jumps in a monotone function of a single variable, sources of pollution along a river, and leaks in a pipeline. The fact that this elegant combinatorial algorithm has influenced so many diverse areas underscores its importance.

# CHAPTER 2

## CONCEPTS OF COMPRESSION

### 1. Information theory

A code is a mapping of source messages (words from the source alphabet  $\alpha$ ) into codewords (words of the code alphabet  $\beta$ ). The source messages are the basic units into which the string to be represented is partitioned. These basic units may be single symbols from the source alphabet, or they may be strings of symbols. For string EXAMPLE,  $\alpha = \{a, b, c, d, e, f, g, \text{space}\}$ . For purpose of explanation, let  $\beta$  will be taken to be  $\{0, 1\}$ . Codes can be categorized as block-block, block-variable, variable-block or variable-variable, where block-block indicates that the source messages and codewords are of fixed length and variable-variable codes map variable-length source messages into variable-length codewords. A block-block code for EXAMPLE is shown in Figure 1.1 and a variable-variable code is given in Figure 1.2. If the string EXAMPLE were coded using the Figure 1.1 code, the length of the coded message would be 120; using Figure 1.2 the length would be 30.

<i>source message</i>	<i>codeword</i>	<i>source message</i>	<i>codeword</i>
<i>a</i>	000	<i>aa</i>	0
<i>b</i>	001	<i>bbb</i>	1
<i>c</i>	010	<i>cccc</i>	10
<i>d</i>	011	<i>dddd</i>	11
<i>e</i>	100	<i>eeeeee</i>	100
<i>f</i>	101	<i>ffffff</i>	101
<i>g</i>	110	<i>gggggggg</i>	110
<i>space</i>	111	<i>space</i>	111

Figure 1.1: A block-block code    Figure 1.2: A variable-variable code.

The oldest and most widely used codes, ASCII and EBCDIC, are examples of block-block codes, mapping an alphabet of 64 (or 256) single characters onto 6-bit (or 8-bit) codewords. These are not discussed, as they do not provide compression. The codes featured in this survey are of the block-variable, variable-variable, and variable-block types.

When source messages of variable length are allowed, the question of how a message ensemble (sequence of messages) is parsed into individual messages arises. Many of the algorithms described here are defined-word schemes. That is, the set of source messages is determined prior to the invocation of the coding scheme. For example, in text file processing each character may constitute a message, or messages may be defined to consist of alphanumeric and non-alphanumeric strings. In Pascal source code, each token may represent a message. All codes involving fixed-length source messages are, by default, defined-word codes. In free-parse methods, the coding algorithm itself parses the ensemble into variable-length sequences of symbols. Most of the known data compression methods are defined-word schemes; the free-parse model differs in a fundamental way from the classical coding paradigm.

A code is distinct if each codeword is distinguishable from every other (i.e., the mapping from source messages to codewords is one-to-one). A distinct code is uniquely decodable if every codeword is identifiable when immersed in a sequence of codewords. Clearly, each of these features is desirable. The codes of Figure 1.1 and Figure 1.2 are both distinct, but the code of Figure 1.2 is not uniquely decodable. For example, the coded message 11 could be decoded as either ddddd or bbbbbb. A uniquely decodable code is a prefix code (or prefix-free code) if it has the prefix property, which requires that no codeword is a proper prefix of any other codeword. All uniquely decodable block-block and variable-block codes are prefix codes. The code with codewords  $\{1, 100000, 00\}$  is an example of a code which is uniquely decodable but which does not have the prefix property. Prefix codes are instantaneously decodable; that is, they have the desirable property that the coded message can be parsed into codewords without the need for lookahead. In order to decode a message encoded using the codeword set  $\{1, 100000, 00\}$ , lookahead is required. For example, the first codeword of the message 1000000001 is 1, but this cannot be determined until the last (tenth) symbol of the message is read (if the string of zeros had been of odd length, then the first codeword would have been 100000).

A minimal prefix code is a prefix code such that if  $x$  is a proper prefix of some codeword, then  $x\sigma$  is either a codeword or a proper prefix of a codeword, for each letter  $\sigma$  in  $\beta$ . The set of codewords  $\{00, 01, 10\}$  is an example of a prefix code which is not minimal. The fact that 1 is a proper prefix of the codeword 10 requires

that it be either a codeword or a proper prefix of a codeword, and it is neither. Intuitively, the minimality constraint prevents the use of codewords which are longer than necessary. In the above example the codeword 10 could be replaced by the codeword 1, yielding a minimal prefix code with shorter codewords. The codes discussed in this paper are all minimal prefix codes.

In this section, a code has been defined to be a mapping from a source alphabet to a code alphabet; we now define related terms. The process of transforming a source ensemble into a coded message is coding or encoding. The encoded message may be referred to as an encoding of the source ensemble. The algorithm which constructs the mapping and uses it to transform the source ensemble is called the encoder. The decoder performs the inverse operation, restoring the coded message to its original form.

## **2. A Data Compression Model**

In order to discuss the relative merits of data compression techniques, a framework for comparison must be established. There are two dimensions along which each of the schemes discussed here may be measured, algorithm complexity and amount of compression. When data compression is used in a data transmission application, the goal is speed.

Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message, and the time required for the decoder to recover the original ensemble. In a data storage application, although the degree of compression is the primary concern, it is nonetheless necessary that the algorithm be efficient in order for the scheme to be practical. For a static scheme, there are three algorithms to analyze: the map construction algorithm, the encoding algorithm, and the decoding algorithm. For a dynamic scheme, there are just two algorithms: the encoding algorithm, and the decoding algorithm.

Several common measures of compression have been suggested: redundancy [Shannon and Weaver 1949], average message length [Huffman 1952], and compression ratio [Rubin 1976; Ruth and Kreutzer 1972]. These measures are defined below.

Related to each of these measures are assumptions about the characteristics of the source. It is generally assumed in information theory that all statistical parameters of a message source are known with perfect accuracy [Gilbert 1971].

The most common model is that of a discrete memoryless source; a source whose output is a sequence of letters (or messages), each letter being a selection from some fixed alphabet  $a, \dots$ . The letters are taken to be random, statistically independent selections from the alphabet, the selection being made according to some fixed probability assignment  $p(a), \dots$  [Gallager 1968]. To avoid loss of generality, the code alphabet is assumed to be  $\{0,1\}$  throughout most papers. The modifications necessary for larger code alphabets are straightforward.

It is assumed that any cost associated with the code letters is uniform. This is a reasonable assumption, although it omits applications like telegraphy where the code symbols are of different durations. The assumption is also important, since the problem of constructing optimal codes over unequal code letter costs is a significantly different and more difficult problem.

Perl et al. and Varn have developed algorithms for minimum-redundancy prefix coding in the case of arbitrary symbol cost and equal codeword probability [Perl et al. 1975; Varn 1971]. The assumption of equal probabilities mitigates the difficulty presented by the variable symbol cost. For the more general unequal letter costs and unequal probabilities model, Karp has proposed an integer linear programming approach [Karp 1961]. There have been several approximation algorithms proposed for this more difficult problem [Krause 1962; Cot 1977; Mehlhorn 1980].

When data is compressed, the goal is to reduce redundancy, leaving only the informational content. The measure of information of a source message  $x$  (in bits) is  $-\lg p(x)$  [ $\lg$  denotes the base 2 logarithm]. This definition has intuitive appeal; in the case that  $p(x)=1$ , it is clear that  $x$  is not at all informative since it had to occur. Similarly, the smaller the value of  $p(x)$ , the more unlikely  $x$  is to appear, hence the larger its information content.

The reader is referred to Abramson for a longer, more elegant discussion of the legitimacy of this technical definition of the concept of information [Abramson 1963, pp. 6-13]. The average information content over the source alphabet can be computed by weighting the information content of each source letter by its probability of occurrence,



yielding the expression  $\sum_{i=1}^n [-p(a(i)) \lg p(a(i))]$ . This quantity is referred to as the *entropy* of a source letter, or the entropy of the source, and is denoted by  $H$ .

Since the length of a codeword for message  $a(i)$  must be sufficient to carry the information content of  $a(i)$ , entropy imposes a lower bound on the number of bits required for the coded message. The total number of bits must be at least as large as the product of  $H$  and the length of the source ensemble. Since the value of  $H$  is generally not an integer, variable length codewords must be used if the lower bound is to be achieved. Given that message *EXAMPLE* is to be encoded one letter at a time, the entropy of its source can be calculated using a standard set of probabilities:  $H = 2.894$ , so that the minimum number of bits contained in an encoding of *EXAMPLE* is 116.

The Huffman code does not quite achieve the theoretical minimum in this case.

Both of these definitions of information content are due to Shannon. A derivation of the concept of entropy as it relates to information theory is presented by Shannon [Shannon and Weaver 1949]. A simpler, more intuitive explanation of entropy is offered by Ash [Ash 1965].

The most common notion of a "good" code is one which is *optimal* in the sense of having minimum redundancy. *Redundancy* can be defined as:  $\sum p(a(i)) l(i) - \sum [-p(a(i)) \lg p(a(i))]$  where  $l(i)$  is the length of the codeword representing message  $a(i)$ . The expression  $\sum p(a(i)) l(i)$  represents the lengths of the codewords weighted by their probabilities of occurrence, that is, the average codeword length. The expression  $\sum [-p(a(i)) \lg p(a(i))]$  is entropy,  $H$ .

Thus, redundancy is a measure of the difference between average codeword length and average information content. If a code has minimum average codeword length for a given discrete probability distribution, it is said to be a minimum redundancy code.

We define the term *local redundancy* to capture the notion of redundancy caused by local properties of a message ensemble, rather than its global characteristics. While the model used for analyzing general-purpose coding techniques assumes a random distribution of the source messages, this may not actually be the case. In particular applications the tendency for messages to cluster in predictable patterns may be known. The existence of predictable patterns may be exploited to minimize local redundancy.

Huffman uses *average message length*,  $\sum p(a(i)) l(i)$ , as a measure of the efficiency of a code. Clearly the meaning of this term is the average length of a *coded* message. We will use the term *average codeword length* to represent this quantity. Since redundancy is defined to be average codeword length minus entropy and entropy is constant for a given probability distribution, minimizing average codeword length minimizes redundancy.

A code is *asymptotically optimal* if it has the property that for a given probability distribution, the ratio of average codeword length to entropy approaches 1 as entropy tends to infinity. That is, asymptotic optimality guarantees that average codeword length approaches the theoretical minimum (entropy represents information content, which imposes a lower bound on codeword length).

The amount of compression yielded by a coding scheme can be measured by a *compression ratio*. The term compression ratio has been defined in several ways. The definition  $C = (\text{average message length})/(\text{average codeword length})$  captures the common meaning, which is a comparison of the length of the coded message to the length of the original ensemble [Cappellini 1985].

If we think of the characters of the ensemble *EXAMPLE* as 6-bit ASCII characters, then the average message length is 6 bits. The Huffman code represents *EXAMPLE* in 117 bits in a certain representation, or 2.9 bits per character. This yields a compression ratio of  $6/2.9$ , representing compression by a factor of more than 2. Alternatively, we may say that Huffman encoding produces a file whose size is 49% of the original ASCII file, or that 49% compression has been achieved.

A somewhat different definition of compression ratio, by Rubin,  $C = (S - O - OR)/S$ , includes the representation of the code itself in the transmission cost [Rubin 1976]. In this definition  $S$  represents the length of the source ensemble,  $O$  the length of the output (coded message), and  $OR$  the size of the "output representation" (e.g., the number of bits required for the encoder to transmit the code mapping to the decoder). The quantity  $OR$  constitutes a "charge" to an algorithm for transmission of information about the coding scheme. The intention is to measure the total size of the transmission (or file to be stored).

### 3. Semantic Dependent Methods

Semantic dependent data compression techniques are designed to respond to specific types of local redundancy occurring in certain applications. One area in which data compression is of great importance is image representation and processing. There are two major reasons for this. The first is that digitized images contain a large amount of local redundancy. An image is usually captured in the form of an array of pixels whereas methods which exploit the tendency for pixels of like color or intensity to cluster together may be more efficient.

The second reason for the abundance of research in this area is volume. Digital images usually require a very large number of bits, and many uses of digital images involve large collections of images.

One technique used for compression of image data is *run length encoding*. In a common version of run length encoding, the sequence of image elements along a scan line (row) is mapped into a sequence of pairs  $(c,l)$  where  $c$  represents an intensity or color and  $l$  the length of the run (sequence of pixels of equal intensity). For pictures such as weather maps, run length encoding can save a significant number of bits over the image element sequence [Gonzalez and Wintz 1977].

Another data compression technique specific to the area of image data is *difference mapping*, in which the image is represented as an array of differences in brightness (or color) between adjacent pixels rather than the brightness values themselves. Difference mapping was used to encode the pictures of Uranus transmitted by *Voyager 2*.

The 8 bits per pixel needed to represent 256 brightness levels was reduced to an average of 3 bits per pixel when difference values were transmitted [Laeser et al. 1986]. In spacecraft applications, image fidelity is a major concern due to the effect of the distance from the spacecraft to earth on transmission reliability. Difference mapping was combined with error-correcting codes to provide both compression and data integrity in the *Voyager* project.

Another method which takes advantage of the tendency for images to contain large areas of constant intensity is the use of the quadtree data structure [Samet 1984]. Additional examples of coding techniques used in image processing can be found in Wilkins and Wintz and in Cappellini [Wilkins and Wintz 1971; Cappellini 1985].

Data compression is of interest in business data processing, both because of the cost savings it offers and because of the large volume of data manipulated in many business applications. The types of local redundancy present in business data files include runs of zeros in numeric fields, sequences of blanks in alphanumeric fields, and fields which are present in some records and null in others. Run length encoding can be used to compress sequences of zeros or blanks. Null suppression may be accomplished through the use of presence bits [Ruth and Kreutzer 1972].

Another class of methods exploits cases in which only a limited set of attribute values exist. *Dictionary substitution* entails replacing alphanumeric representations of information such as bank account type, insurance policy type, sex, month, etc. by the few bits necessary to represent the limited number of possible attribute values [Reghbati 1981].

### **3.1 IBM's Information Management System**

Cormack describes a data compression system which is designed for use with database files [Cormack 1985]. The method, which is part of IBM's "Information Management System" (IMS), compresses individual records and is invoked each time a record is stored in the database file; expansion is performed each time a record is retrieved.

Since records may be retrieved in any order, context information used by the compression routine is limited to a single record. In order for the routine to be applicable to any database, it must be able to adapt to the format of the record. The fact that database records are usually heterogeneous collections of small fields indicates that the local properties of the data are more important than its global characteristics.

The compression routine in IMS is a hybrid method which attacks this local redundancy by using different coding schemes for different types of fields. The identified field types in IMS are *letters of the alphabet*, *numeric digits*, *packed decimal digit pairs*, *blank*, and *other*.

When compression begins, a default code is used to encode the first character of the record. For each subsequent character, the type of the previous character determines the

code to be used. For example, if the record 01870\_ABCD\_\_LMN were encoded with the *letter* code as default, the leading zero would be coded using the *letter* code; the 1, 8, 7, 0 and the first blank (\_\_) would be coded by the *numeric* code.

The *A* would be coded by the *blank* code; *B*, *C*, *D*, and the next blank by the *letter* code; the next blank and the *L* by the *blank* code; and the *M* and *N* by the *letter* code. Clearly, each code must define a codeword for every character; the *letter* code would assign the shortest codewords to letters, the numeric code would favor the digits, etc. In the system Cormack describes, the types of the characters are stored in the encode/decode data structures.

When a character *c* is received, the decoder checks *type(c)* to detect which code table will be used in transmitting the next character. The compression algorithm might be more efficient if a special bit string were used to alert the receiver to a change in code table. Particularly if fields were reasonably long, decoding would be more rapid and the extra bits in the transmission would not be excessive. Cormack reports that the performance of the IMS compression routines is very good; at least fifty sites are currently using the system. He cites a case of a database containing student records whose size was reduced by 42.1%, and as a side effect the number of disk operations required to load the database was reduced by 32.7% [Cormack 1985].

A variety of approaches to data compression designed with text files in mind include use of a dictionary either representing all of the words in the file so that the file itself is coded as a list of pointers to the dictionary [Hahn 1974], or representing common words and word endings so that the file consists of pointers to the dictionary and encodings of the less common words [Tropper 1982]. Hand-selection of common phrases [Wagner 1973], programmed selection of prefixes and suffixes [Fraenkel et al. 1983] and programmed selection of common character pairs [Snyderman and Hunt 1970; Cortesi 1982] have also been investigated.

This discussion of semantic dependent data compression techniques represents a limited sample of a very large body of research. These methods and others of a like nature are interesting and of great value in their intended domains.

Their obvious drawback lies in their limited utility. It should be noted, however, that much of the efficiency gained through the use of semantic dependent techniques can be achieved through more general methods, albeit to a lesser degree. For example, the dictionary approaches can be implemented through either Huffman coding or Lempel-Ziv

codes. Cormack's database scheme is a special case of the codebook approach and run length encoding is one of the effects of Lempel-Ziv codes.

# CHAPTER 3

## VARIOUS COMPRESSION ALGORITHMS

### 1. Shannon-Fano Coding

The classic defined-word scheme was developed over 30 years ago in Huffman's well-known paper on minimum-redundancy coding [Huffman 1952]. Huffman's algorithm provided the first solution to the problem of constructing minimum-redundancy codes. Many people believe that Huffman coding cannot be improved upon, that is, that it is guaranteed to achieve the best possible compression ratio. This is only true, however, under the constraints that each source message is mapped to a unique codeword and that the compressed text is the concatenation of the codewords for the source messages. An earlier algorithm, due independently to Shannon and Fano [Shannon and Weaver 1949; Fano 1949], is not guaranteed to provide optimal codes, but approaches optimal behavior as the number of messages approaches infinity. The Huffman algorithm is also of importance because it has provided a foundation upon which other data compression techniques have built and a benchmark to which they may be compared. We classify the codes generated by the Huffman and Shannon-Fano algorithms as variable-variable and note that they include block-variable codes as a special case, depending upon how the source messages are defined.

The Shannon-Fano technique has as an advantage its simplicity. The code is constructed as follows: the source messages  $a(i)$  and their probabilities  $p(a(i))$  are listed in order of nonincreasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have codewords beginning with 1. Each of these groups is then divided according to the same criterion and additional code digits are appended. The process is continued until each subset contains only one message. Clearly the Shannon-Fano algorithm yields a minimal prefix code.

a	1/2	0
b	1/4	10
c	1/8	110
d	1/16	1110
e	1/32	11110
f	1/32	11111

*Figure 3.1 A Shannon-Fano Code*

Figure 3.1 shows the application of the method to a particularly simple probability distribution. The length of each codeword  $x$  is equal to  $-\lg p(x)$ . This is true as long as it is possible to divide the list into subgroups of exactly equal probability. When this is not possible, some codewords may be of length  $-\lg p(x)+1$ . The Shannon-Fano algorithm yields an average codeword length  $S$  which satisfies  $H \leq S \leq H + 1$ . In Figure 3.2, the Shannon-Fano code for ensemble *EXAMPLE* is given. As is often the case, the average codeword length is the same as that achieved by the Huffman code. That the Shannon-Fano algorithm is not guaranteed to produce an optimal code is demonstrated by the following set of probabilities: { .35, .17, .17, .16, .15 }.

G	8/40	00
f	7/40	010
e	6/40	011
d	5/16	100
space	5/32	101
c	4/32	110
b	3/40	1110
a	2/40	1111

*Figure 3.2A Shannon-Fano Code for EXAMPLE*

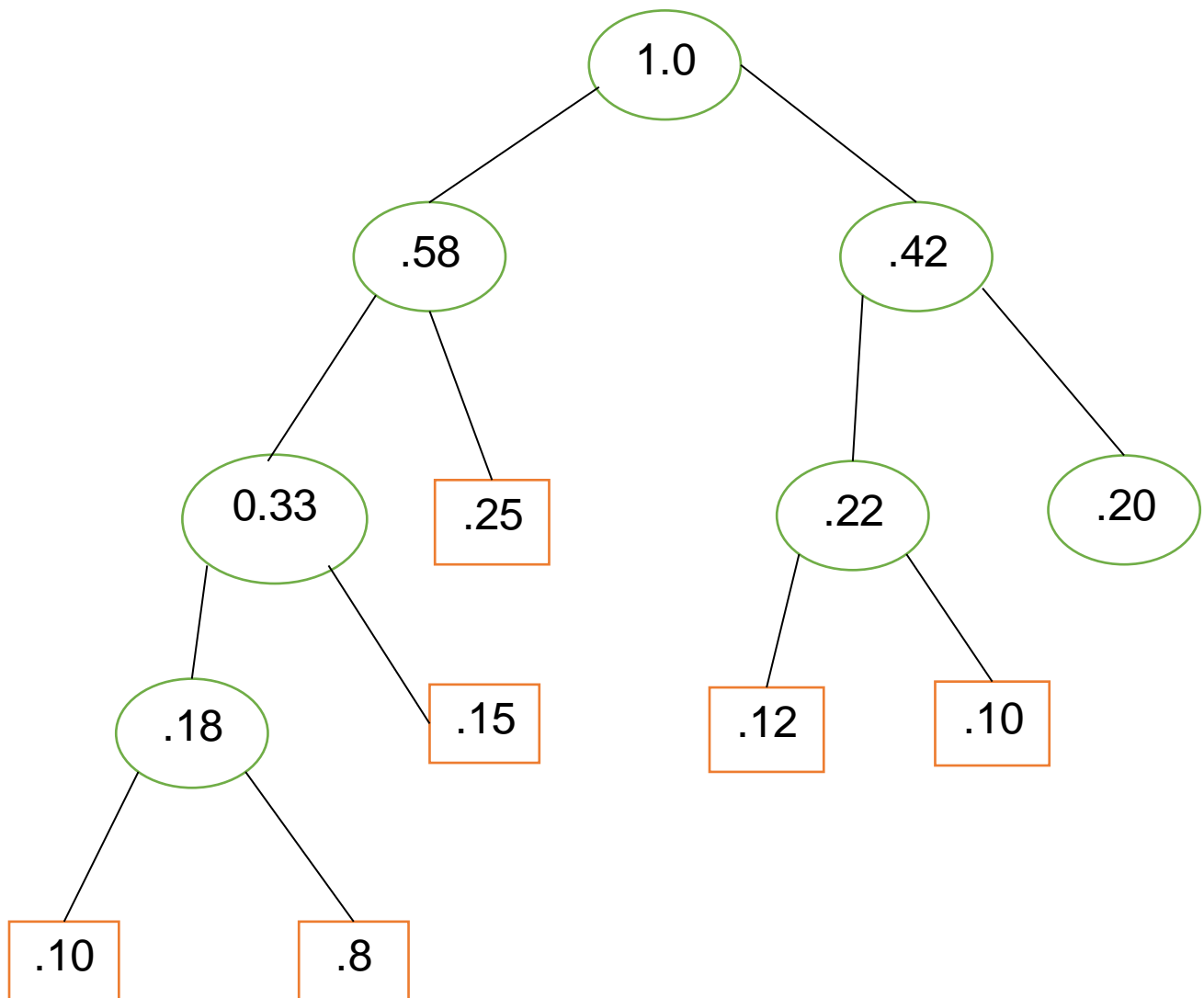


## 2. Static Huffman Coding

Huffman's algorithm, expressed graphically, takes as input a list of nonnegative weights  $\{w(1), \dots, w(n)\}$  and constructs a full binary tree [a binary tree is full if every node has either zero or two children] whose leaves are labeled with the weights. When the Huffman algorithm is used to construct a code, the weights represent the probabilities associated with the source letters. Initially there is a set of singleton trees, one for each weight in the list. At each step in the algorithm the trees corresponding to the two smallest weights,  $w(i)$  and  $w(j)$ , are merged into a new tree whose weight is  $w(i)+w(j)$  and whose root has two children which are the subtrees represented by  $w(i)$  and  $w(j)$ . The weights  $w(i)$  and  $w(j)$  are removed from the list and  $w(i)+w(j)$  is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen. In Huffman's paper, the process begins with a non-increasing list of weights. This detail is not important to the correctness of the algorithm, but it does provide a more efficient implementation [Huffman 1952]. The Huffman algorithm is demonstrated in Figure 3.3

a1	.25	.25	.25	.33	.42	.58	1.0
a2	.20	.20	.22	.25	.33	.42	
a3	.15	.18	.20	.22	.25		
a4	.12	.15	.18	.20			
a5	.10	.12	.15				
a6	.10	.10					
a7	.08						

*Figure 3.3 Huffman Process – The List*



*Fig 3.3 Huffman Process – The Tree*

The Huffman algorithm determines the lengths of the codewords to be mapped to each of the source letters  $a(i)$ . There are many alternatives for specifying the actual digits; it is necessary only that the code have the prefix property. The usual assignment entails labeling the edge from each parent to its left child with the digit 0 and the edge to the right child with 1. The codeword for each source letter is the sequence of labels along the path from the root to the leaf node representing that letter. The codewords for the source of Figure 3.3, in order of decreasing probability, are {01,11,001,100,101,000,0001}.

Clearly, this process yields a minimal prefix code. Further, the algorithm is guaranteed to produce an optimal (minimum redundancy) code [Huffman 1952].

Gallager has proved an upper bound on the redundancy of a Huffman code of  $p(n) + \lg[(2 \lg e)/e]$  which is approximately  $p(n) + 0.086$ , where  $p(n)$  is the probability of the least likely source message [Gallager 1978]. In a recent paper, Capocelli et al. provide new bounds which are tighter than those of Gallager for some probability distributions [Capocelli et al. 1986]. Figure 3.4 shows a distribution for which the Huffman code is optimal while the Shannon-Fano code is not.

In addition to the fact that there are many ways of forming codewords of appropriate lengths, there are cases in which the Huffman algorithm does not uniquely determine these lengths due to the arbitrary choice among equal minimum weights. As an example, codes with codeword lengths of  $\{1,2,3,4,4\}$  and of  $\{2,2,2,3,3\}$  both yield the same average codeword length for a source with probabilities  $\{.4,.2,.2,.1,.1\}$ .

Schwartz defines a variation of the Huffman algorithm which performs "bottom merging"; that is, orders a new parent node above existing nodes of the same weight and always merges the last two weights in the list. The code constructed is the Huffman code with minimum values of maximum codeword length ( $\text{MAX}\{l(i)\}$ ) and total codeword length ( $\text{SUM}\{l(i)\}$ ) [Schwartz 1964]. Schwartz and Kallick describe an implementation of Huffman's algorithm with bottom merging [Schwartz and Kallick 1964].

The Schwartz-Kallick algorithm and a later algorithm by Connell [Connell 1973] use Huffman's procedure to determine the lengths of the codewords, and actual digits are assigned so that the code has the numerical sequence property. That is, codewords of equal length form a consecutive sequence of binary numbers. Shannon-Fano codes also have the numerical sequence property. This property can be exploited to achieve a compact representation of the code and rapid encoding and decoding.

		<i>S-F</i>	<i>Huffman</i>
a(1)	.35	00	1
a(2)	.17	01	011
a(3)	.17	10	010
a(4)	.16	110	001
a(5)	.15	111	000
<i>Average Codeword length</i>		2.31	2.30

*Fig 3.4 Comparison of Shannon-Fano and Huffman Codes*

Both the Huffman and the Shannon-Fano mappings can be generated in  $O(n)$  time, where  $n$  is the number of messages in the source ensemble (assuming that the weights have been presorted). Each of these algorithms maps a source message  $a(i)$  with probability  $p$  to a codeword of length  $l$  ( $-\lg p \leq l \leq -\lg p + 1$ ). Encoding and decoding times depend upon the representation of the mapping. If the mapping is stored as a binary tree, then decoding the codeword for  $a(i)$  involves following a path of length  $l$  in the tree.

A table indexed by the source messages could be used for encoding; the code for  $a(i)$  would be stored in position  $i$  of the table and encoding time would be  $O(l)$ . Connell's algorithm makes use of the *index* of the Huffman code, a representation of the distribution of codeword lengths, to encode and decode in  $O(c)$  time where  $c$  is the number of different codeword lengths. Tanaka presents an implementation of Huffman coding based on finite-state machines which can be realized efficiently in either hardware or software [Tanaka 1987].

As noted earlier, the redundancy bound for Shannon-Fano codes is 1 and the bound for the Huffman method is  $p(n) + 0.086$  where  $p(n)$  is the probability of the least likely source message (so  $p(n)$  is less than or equal to .5, and generally much less). It is important to note that in defining redundancy to be average codeword length minus entropy, the cost of transmitting the code mapping computed by these algorithms is ignored. The overhead cost for any method where the source alphabet has not been established prior to transmission includes  $n \lg n$  bits for sending the  $n$  source letters. For a

Shannon-Fano code, a list of codewords ordered so as to correspond to the source letters could be transmitted. The additional time required is then  $\text{SUM } l(i)$ , where the  $l(i)$  are the lengths of the codewords. For Huffman coding, an encoding of the shape of the code tree might be transmitted. Since any full binary tree may be a legal Huffman code tree, encoding tree shape may require as many as  $\lg 4^n = 2n$  bits. In most cases the message ensemble is very large, so that the number of bits of overhead is minute by comparison to the total length of the encoded transmission. However, it is imprudent to ignore this cost.

If a less-than-optimal code is acceptable, the overhead costs can be avoided through a prior agreement by sender and receiver as to the code mapping. Rather than using a Huffman code based upon the characteristics of the current message ensemble, the code used could be based on statistics for a class of transmissions to which the current ensemble is assumed to belong. That is, both sender and receiver could have access to a *codebook* with  $k$  mappings in it; one for Pascal source, one for English text, etc.

The sender would then simply alert the receiver as to which of the common codes he is using. This requires only  $\lg k$  bits of overhead. Assuming that classes of transmission with relatively stable characteristics could be identified, this hybrid approach would greatly reduce the redundancy due to overhead without significantly increasing expected codeword length. In addition, the cost of computing the mapping would be amortized over all files of a given class.

That is, the mapping would be computed once on a statistically significant sample and then used on a great number of files for which the sample is representative. There is clearly a substantial risk associated with assumptions about file characteristics and great care would be necessary in choosing both the sample from which the mapping is to be derived and the categories into which to partition transmissions. An extreme example of the risk associated with the codebook approach is provided by author Ernest V. Wright who wrote a novel *Gadsby* (1939) containing no occurrences of the letter E. Since E is the most commonly used letter in the English language, an encoding based upon a sample from *Gadsby* would be disastrous if used with "normal" examples of English text. Similarly, the "normal" encoding would provide poor compression of *Gadsby*.

McIntyre and Pechura describe an experiment in which the codebook approach is compared to static Huffman coding [McIntyre and Pechura 1985]. The sample used for

comparison is a collection of 530 source programs in four languages. The codebook contains a Pascal code tree, a FORTRAN code tree, a COBOL code tree, a PL/1 code tree, and an ALL code tree.

The Pascal code tree is the result of applying the static Huffman algorithm to the combined character frequencies of all of the Pascal programs in the sample. The ALL code tree is based upon the combined character frequencies for all of the programs. The experiment involves encoding each of the programs using the five codes in the codebook and the static Huffman algorithm. The data reported for each of the 530 programs consists of the size of the coded program for each of the five predetermined codes, and the size of the coded program plus the size of the mapping (in table form) for the static Huffman method.

In every case, the code tree for the language class to which the program belongs generates the most compact encoding. Although using the Huffman algorithm on the program itself yields an optimal mapping, the overhead cost is greater than the added redundancy incurred by the less-than-optimal code. In many cases, the ALL code tree also generates a more compact encoding than the static Huffman algorithm. In the worst case, an encoding constructed from the codebook is only 6.6% larger than that constructed by the Huffman algorithm. These results suggest that, for files of source code, the codebook approach may be appropriate.

Gilbert discusses the construction of Huffman codes based on inaccurate source probabilities [Gilbert 1971]. A simple solution to the problem of incomplete knowledge of the source is to avoid long codewords, thereby minimizing the error of underestimating badly the probability of a message. The problem becomes one of constructing the optimal binary tree subject to a height restriction ([Knuth 1971; Hu and Tan 1972; Garey 1974]). Another approach involves collecting statistics for several sources and then constructing a code based upon some combined criterion.

This approach could be applied to the problem of designing a single code for use with English, French, German, etc., sources. To accomplish this, Huffman's algorithm could be used to minimize either the average codeword length for the combined source probabilities; or the average codeword length for English, subject to constraints on average codeword lengths for the other sources.

## CHAPTER 4

### ADAPTIVE HUFFMAN CODING

Adaptive Huffman coding was first conceived independently by Faller and Gallager [Faller 1973; Gallager 1978]. Knuth contributed improvements to the original algorithm [Knuth 1985] and the resulting algorithm is referred to as algorithm FGK. A more recent version of adaptive Huffman coding is described by Vitter [Vitter 1987].

All of these methods are defined-word schemes which determine the mapping from source messages to codewords based upon a running estimate of the source message probabilities. The code is adaptive, changing so as to remain optimal for the current estimates. In this way, the adaptive Huffman codes respond to locality. In essence, the encoder is "learning" the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree so as to stay in synchronization with the encoder.

Another advantage of these systems is that they require only one pass over the data. Of course, one-pass methods are not very interesting if the number of bits they transmit is significantly greater than that of the two-pass scheme. Interestingly, the performance of these methods, in terms of number of bits transmitted, can be better than that of static Huffman coding. This does not contradict the optimality of the static method as the static method is optimal only over all methods which assume a time-invariant mapping.

The performance of the adaptive methods can also be worse than that of the static method. Upper bounds on the redundancy of these methods are presented in this section. As discussed in the introduction, the adaptive method of Faller, Gallager and Knuth is the basis for the UNIX utility compact. The performance of compact is quite good, providing typical compression factors of 30-40%.

## 1. FGK Algorithm

The basis for algorithm FGK is the Sibling Property, defined by Gallager [Gallager 1978]: A binary code tree has the sibling property if each node (except the root) has a sibling and if the nodes can be listed in order of non-increasing weight with each node adjacent to its sibling. Gallager proves that a binary prefix code is a Huffman code if and only if the code tree has the sibling property. In algorithm FGK, both sender and receiver maintain dynamically changing Huffman code trees. The leaves of the code tree represent the source messages and the weights of the leaves represent frequency counts for the messages. At any point in time,  $k$  of the  $n$  possible source messages have occurred in the message ensemble.

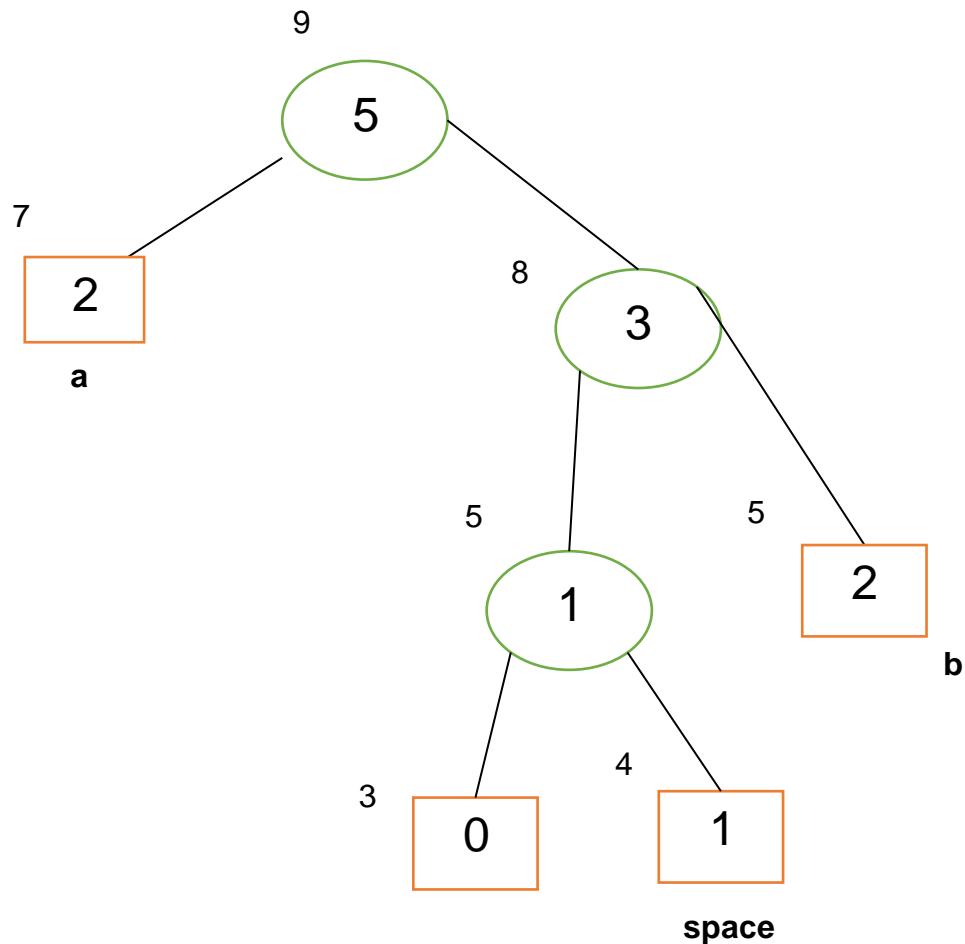


Fig 4.1(a) Tree after processing "aa bb"; 11 will be transmitted for the next  $b$ .



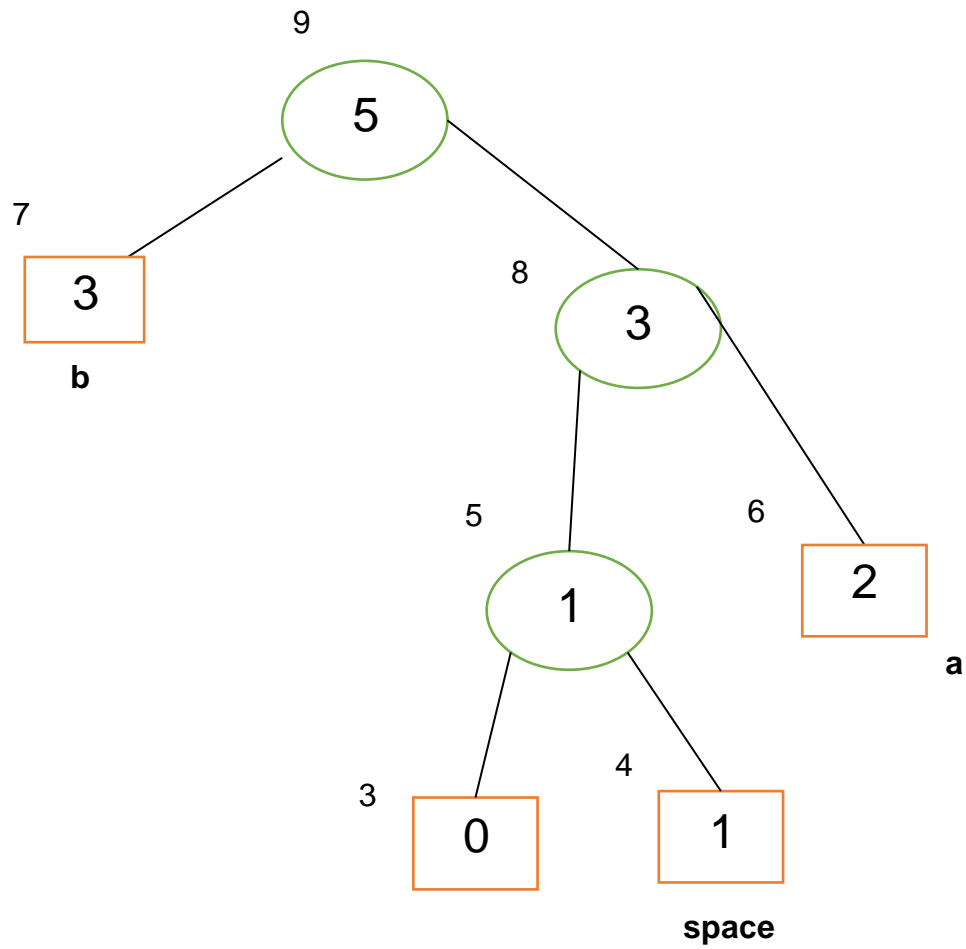


Fig 4.1(b) After encoding the third *b*; 101 will be transmitted for the next *space*; the tree will not change; 100 will be transmitted for the first *c*.

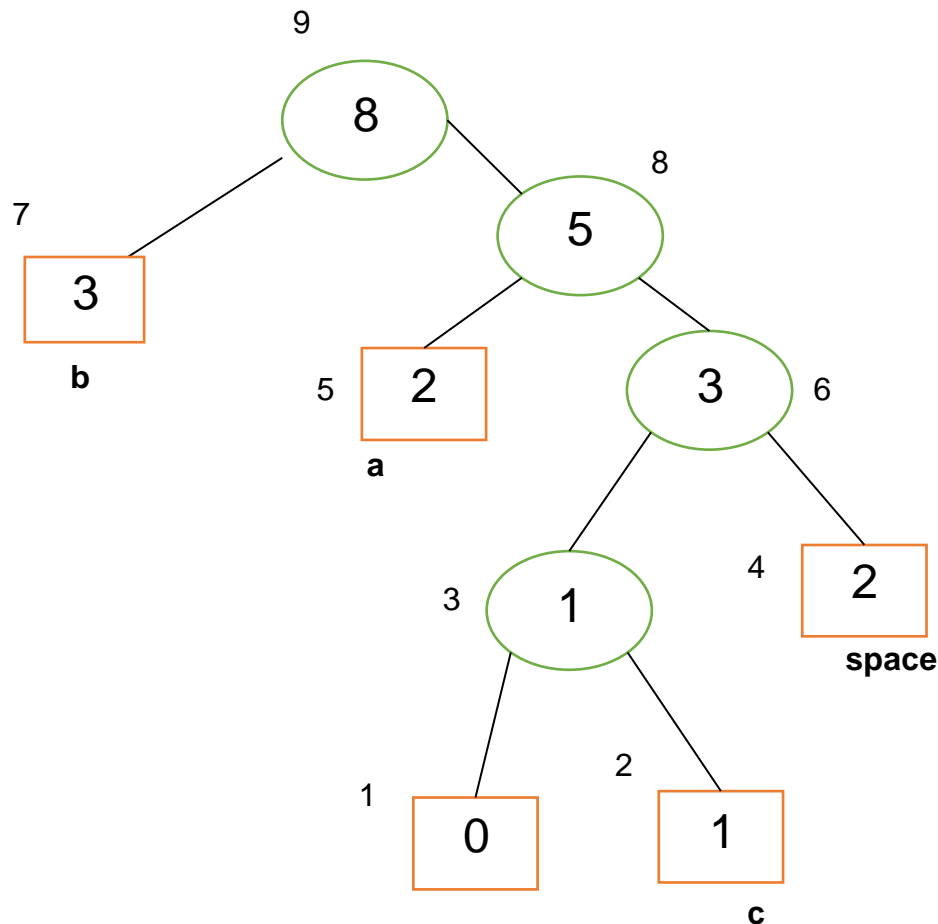


Fig 4.1( c ) Tree after update following first  $c$

Initially, the code tree consists of a single leaf node, called the 0-node. The 0-node is a special node used to represent the  $n-k$  unused messages. For each message transmitted, both parties must increment the corresponding weight and recompute the code tree to maintain the sibling property. At the point in time when  $t$  messages have been transmitted,  $k$  of them distinct, and  $k < n$ , the tree is a legal Huffman code tree with  $k+1$  leaves, one for each of the  $k$  messages and one for the 0-node. If the  $(t+1)$ st message is one of the  $k$  already seen, the algorithm transmits  $a(t+1)$ 's current code, increments the appropriate counter and recomputes the tree. If an unused message occurs, the 0-node is split to create a pair of leaves, one for  $a(t+1)$ , and a sibling which is the new 0-node. Again the tree is recomputed. In this case, the code for the 0-node is sent; in addition, the receiver must be told which of the  $n-k$  unused messages has appeared. At each node a count of occurrences of the corresponding message is stored. Nodes are numbered indicating their position in the sibling property ordering. The updating of the tree can be

done in a single traversal from the  $a(t+1)$  node to the root. This traversal must increment the count for the  $a(t+1)$  node and for each of its ancestors. Nodes may be exchanged to maintain the sibling property, but all of these exchanges involve a node on the path from  $a(t+1)$  to the root. Figure 4.2 shows the final code tree formed by this process on the ensemble *EXAMPLE*.

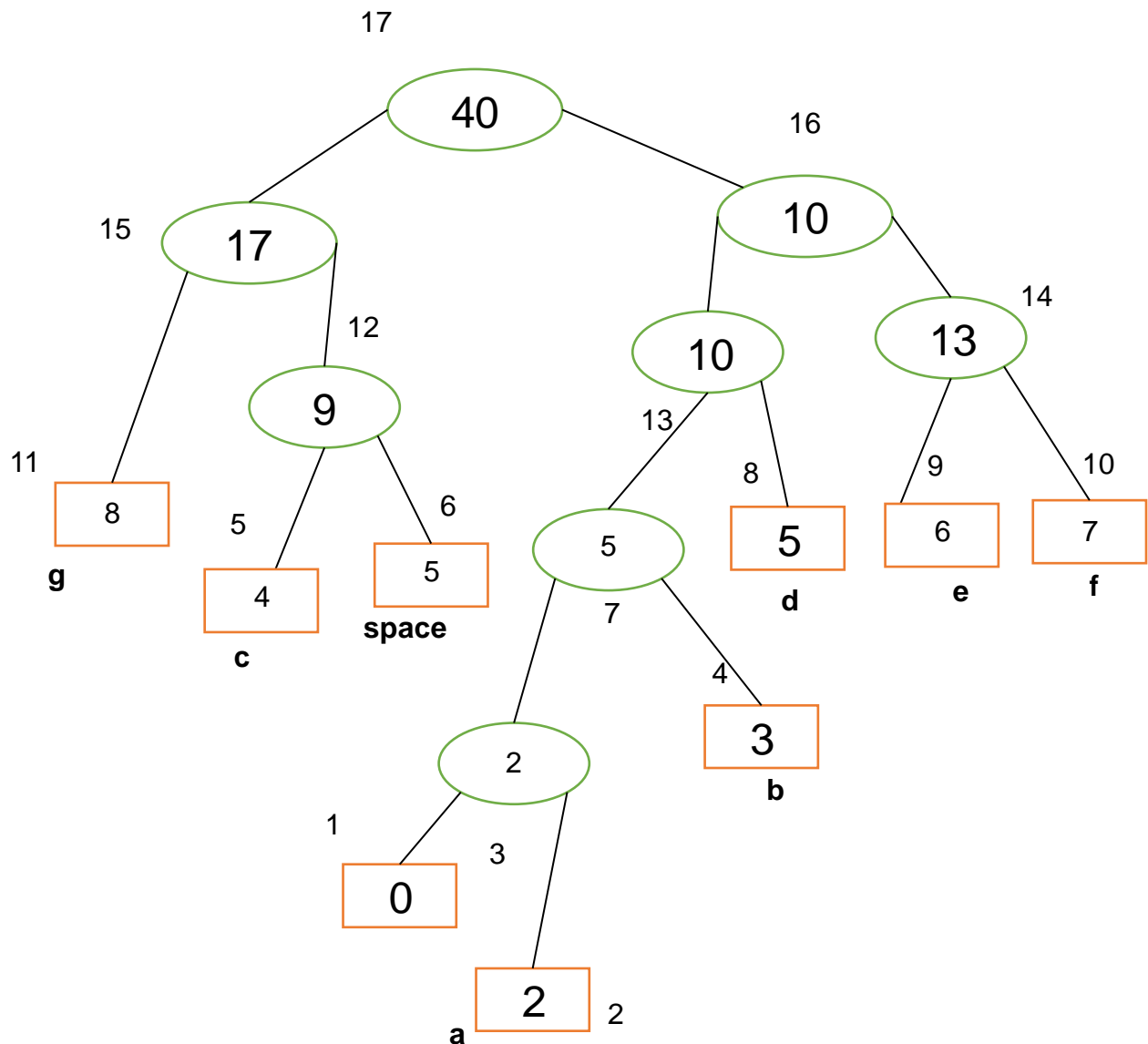


Fig 4.2 Tree formed by algorithm FGK for ensemble *EXAMPLE*

Disregarding overhead, the number of bits transmitted by algorithm FGK for the *EXAMPLE* is 129. The static Huffman algorithm would transmit 117 bits in processing the same data. The overhead associated with the adaptive method is actually less than that of the static algorithm. In the adaptive case the only overhead is

the  $n \lg n$  bits needed to represent each of the  $n$  different source messages when they appear for the first time. (This is in fact conservative; rather than transmitting a unique code for each of the  $n$  source messages, the sender could transmit the message's position in the list of remaining messages and save a few bits in the average case.) In the static case, the source messages need to be sent as does the shape of the code tree. An efficient representation of the tree shape requires  $2n$  bits. Algorithm FGK compares well with static Huffman coding on this ensemble when overhead is taken into account.

## 2. Vitter Algorithm

Vitter has proved that the total number of bits transmitted by algorithm FGK for a message ensemble of length  $t$  containing  $n$  distinct messages is bounded below by  $S - n + 1$ , where  $S$  is the performance of the static method, and bounded above by  $2S + t - 4n + 2$  [Vitter 1987]. So the performance of algorithm FGK is never much worse than twice optimal. Knuth provides a complete implementation of algorithm FGK and a proof that the time required for each encoding or decoding operation is  $O(l)$ , where  $l$  is the current length of the codeword [Knuth 1985]. It should be noted that since the mapping is defined dynamically, during transmission, the encoding and decoding algorithms stand alone; there is no additional algorithm to determine the mapping as in static methods.

The adaptive Huffman algorithm of Vitter (algorithm V) incorporates two improvements over algorithm FGK. First, the number of interchanges in which a node is moved upward in the tree during a recomputation is limited to one. This number is bounded in algorithm FGK only by  $l/2$  where  $l$  is the length of the codeword for  $a(t+1)$  when the recomputation begins. Second, Vitter's method minimizes the values of  $\text{SUM}\{ l(i) \}$  and  $\text{MAX}\{ l(i) \}$  subject to the requirement of minimizing  $\text{SUM}\{ w(i) l(i) \}$ . The intuitive explanation of algorithm V's advantage over algorithm FGK is as follows: as in algorithm FGK, the code tree constructed by algorithm V is the Huffman code tree for the prefix of the ensemble seen so far. The adaptive methods do not assume that the relative frequencies of a prefix represent accurately the symbol probabilities over the entire message. Therefore, the fact that algorithm V guarantees a tree of minimum height (height =  $\text{MAX}\{ l(i) \}$ ) and minimum external path length ( $\text{SUM}\{ l(i) \}$ ) implies that it is better

suiting for coding the next message of the ensemble, given that any of the leaves of the tree may represent that next message.

These improvements are accomplished through the use of a new system for numbering nodes. The numbering, called an implicit numbering, corresponds to a level ordering of the nodes (from bottom to top and left to right).

The following invariant is maintained in Vitter's algorithm: For each weight  $w$ , all leaves of weight  $w$  precede (in the implicit numbering) all internal nodes of weight  $w$ . Vitter proves that this invariant enforces the desired bound on node promotions [Vitter 1987]. The invariant also implements bottom merging, as discussed in Section 3.2, to minimize  $\text{SUM}\{l(i)\}$  and  $\text{MAX}\{l(i)\}$ . The difference between Vitter's method and algorithm FGK is in the way the tree is updated between transmissions. In order to understand the revised update operation, the following definition of a block of nodes is necessary: Blocks are equivalence classes of nodes defined by  $u$  is equivalent to  $v$  iff  $\text{weight}(u) = \text{weight}(v)$  and  $u$  and  $v$  are either both leaves or both internal nodes. The leader of a block is the highest-numbered (in the implicit numbering) node in the block. Blocks are ordered by increasing weight with the convention that a leaf block always precedes an internal block of the same weight. When an exchange of nodes is required to maintain the sibling property, algorithm V requires that the node being promoted be moved to the position currently occupied by the highest-numbered node in the target block.

In Figure 4.5, the Vitter tree corresponding to Figure 4.1c is shown. This is the first point in *EXAMPLE* at which algorithm FGK and algorithm V differ significantly. At this point, the Vitter tree has height 3 and external path length 12 while the FGK tree has height 4 and external path length 14. Algorithm V transmits codeword 001 for the second  $c$ ; FGK transmits 1101. This demonstrates the intuition given earlier that algorithm V is better suited for coding the next message. The Vitter tree corresponding to Figure 4.2, representing the final tree produced in processing *EXAMPLE*, is only different from Figure 4.2 in that the internal node of weight 5 is to the right of both leaf nodes of weight 5. Algorithm V transmits 124 bits in processing *EXAMPLE*, as compared with the 129 bits of algorithm FGK and 117 bits of static Huffman coding. It should be noted that these figures do not include overhead and, as a result, disadvantage the adaptive methods.

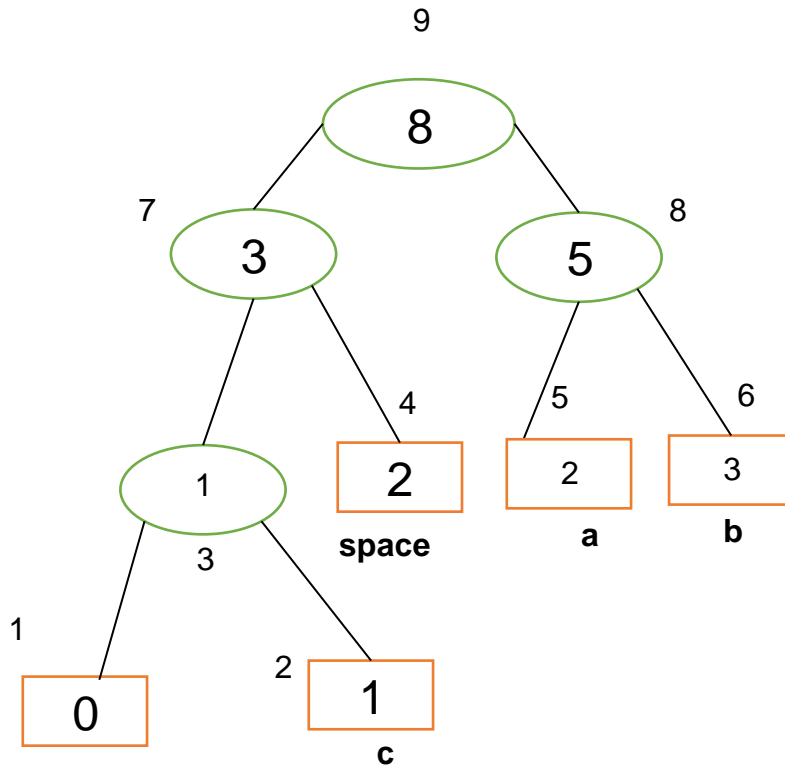


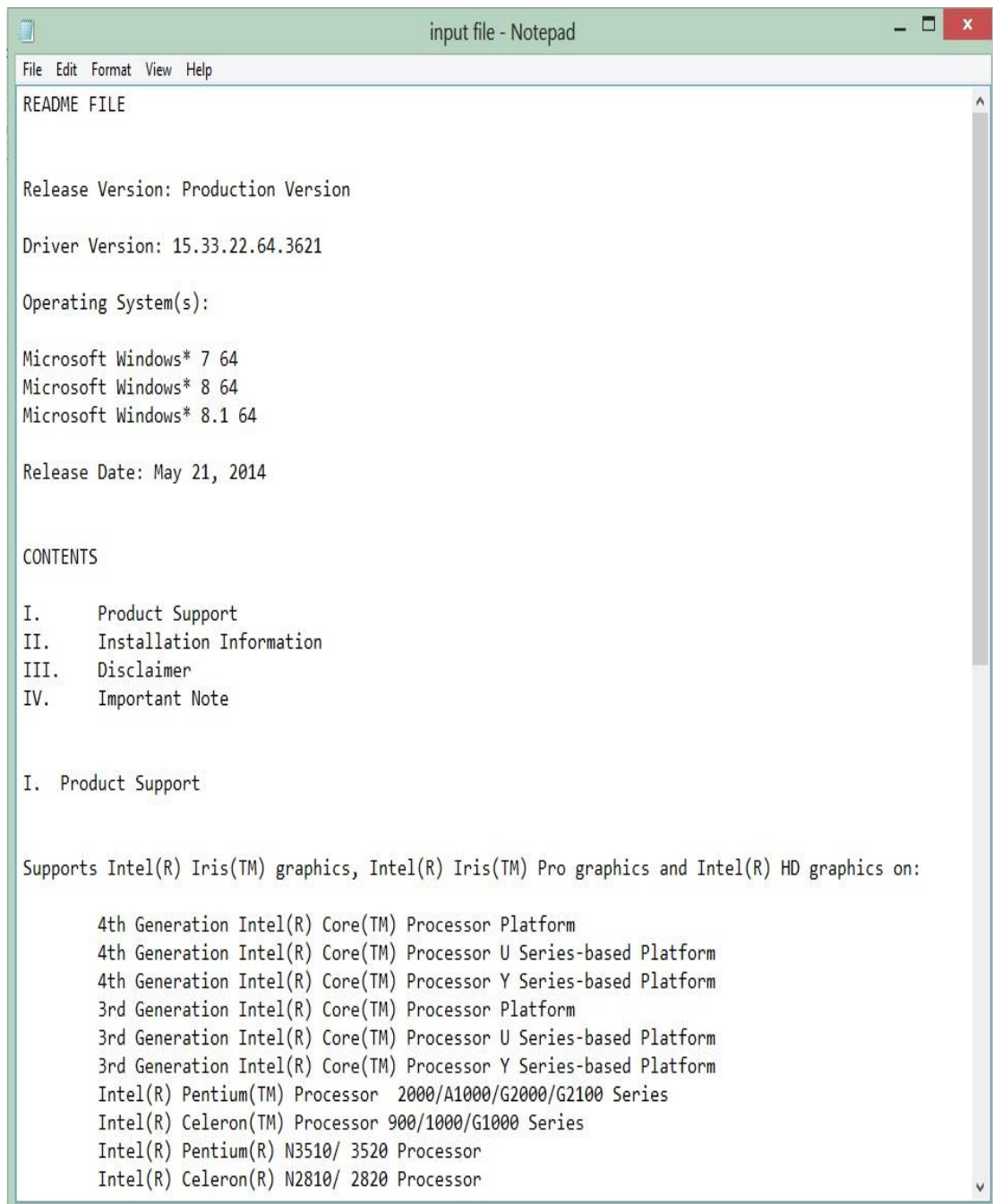
Fig 4.5 Algorithm V processing the ensemble "aabbb c"

It should be noted again that the strategy of minimizing external path length and height is optimal under the assumption that any source letter is equally likely to occur next. Other reasonable strategies include one which assumes locality. To take advantage of locality, the ordering of tree nodes with equal weights could be determined on the basis of recency. Another reasonable assumption about adaptive coding is that the weights in the current tree correspond closely to the probabilities associated with the source. This assumption becomes more reasonable as the length of the ensemble increases. Under this assumption, the expected cost of transmitting the next letter is  $\text{SUM}\{ p(i) l(i) \}$  which is approximately  $\text{SUM}\{ w(i) l(i) \}$ , so that neither algorithm FGK nor algorithm V has any advantage.

Vitter proves that the performance of his algorithm is bounded by  $S - n + 1$  from below and  $S + t - 2n + 1$  from above [Vitter 1987]. At worst then, Vitter's adaptive method may transmit one more bit per codeword than the static Huffman method. The improvements made by Vitter do not change the complexity of the algorithm; algorithm V encodes and decodes in  $O(l)$  time as does algorithm FGK.

### 3. Execution

Input - A readme file - 6 KB in size



```
input file - Notepad
File Edit Format View Help
README FILE

Release Version: Production Version

Driver Version: 15.33.22.64.3621

Operating System(s):

Microsoft Windows* 7 64
Microsoft Windows* 8 64
Microsoft Windows* 8.1 64

Release Date: May 21, 2014

CONTENTS

I.    Product Support
II.   Installation Information
III.  Disclaimer
IV.   Important Note

I. Product Support

Supports Intel(R) Iris(TM) graphics, Intel(R) Iris(TM) Pro graphics and Intel(R) HD graphics on:

    4th Generation Intel(R) Core(TM) Processor Platform
    4th Generation Intel(R) Core(TM) Processor U Series-based Platform
    4th Generation Intel(R) Core(TM) Processor Y Series-based Platform
    3rd Generation Intel(R) Core(TM) Processor Platform
    3rd Generation Intel(R) Core(TM) Processor U Series-based Platform
    3rd Generation Intel(R) Core(TM) Processor Y Series-based Platform
    Intel(R) Pentium(TM) Processor 2000/A1000/G2000/G2100 Series
    Intel(R) Celeron(TM) Processor 900/1000/G1000 Series
    Intel(R) Pentium(R) N3510/ 3520 Processor
    Intel(R) Celeron(R) N2810/ 2820 Processor
```

Program - Adaptive Huffman.exe

```
C:\Users\ARR(0)GaNcE\Documents\C-Free\Temp\adhuff.exe
Release Date: May 21, 2014

CONTENTS
I. Product Support
II. Installation Information
III. Disclaimer
IV. Important Note



I. Product Support

Supports Intel(R) Iris(TM) graphics, Intel
el(R) Iris(TM) Pro graphics and Intel(R)
HD graphics on:

cessor 4th Generation Intel(R) Core(TM) Pro
cessor 4th Generation Intel(R) Core(TM) Pro
cessor 4th Y Series-based Intel(R) Core(TM) Pro
cessor 3rd Generation Intel(R) Core(TM) Pro
cessor 3rd Y Series-based Intel(R) Core(TM) Pro
cessor 3rd Generation Intel(R) Core(TM) Pro
cessor 3rd Y Series-based Intel(R) Core(TM) Pro
cessor Intel(R) Pentium(R) Processor 2000
/ Athlon(R) G2000 / G2100 Series Processor 900/1
000 / G1000 Series
cessor Intel(R) Pentium(R) N3510 / 3520 Proc
cessor Intel(R) Celeron(R) N2810 / 2820
Processor Intel(R) Celeron(R) N29
```



Output : A compressed file (composed of special characters)- 2 KB in size

Name	Date modified	Type	Size
 compressed file	21-12-2014 12:09	Text Document	2 KB
 input file	21-05-2014 17:15	Text Document	6 KB

# CHAPTER 5

## GRAPHS - BACKGROUND

### 1. Introduction

In computer science, a graph is an abstract data type that is meant to implement the graph and directed graph concepts from mathematics.

A graph data structure consists of a finite (and possibly mutable) set of nodes or vertices, together with a set of ordered pairs of these nodes (or, in some cases, a set of unordered pairs). These pairs are known as edges or arcs. As in mathematics, an edge  $(x,y)$  is said to point or go from  $x$  to  $y$ . The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc).

Graphs are mathematical concepts that have found many uses in computer science. Graphs come in many different flavors, many of which have found uses in computer programs. Some flavors are:

- Simple graph
- Undirected or directed graphs
- Cyclic or acyclic graphs
- labeled graphs
- Weighted graphs
- Infinite graphs

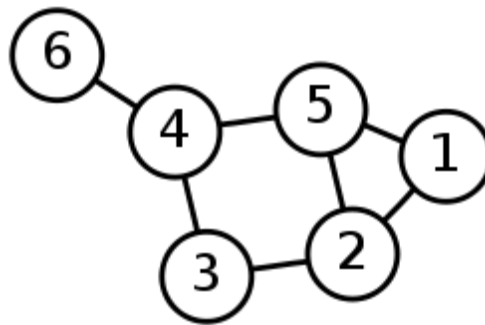
Most graphs are defined as a slight alteration of the following rules.

- A graph is made up of two sets called Vertices and Edges.
- The Vertices are drawn from some underlying type, and the set may be finite or infinite.

- Each element of the Edge set is a pair consisting of two elements from the Vertices set.
- Graphs are often depicted visually, by drawing the elements of the Vertices set as boxes or circles, and drawing the elements of the edge set as lines or arcs between the boxes or circles. There is an arc between  $v_1$  and  $v_2$  if  $(v_1, v_2)$  is an element of the Edge set.

Adjacency : If  $(u, v)$  is in the edge set we say  $u$  is adjacent to  $v$  (which we sometimes write as  $u \sim v$ ).

For example the graph drawn below:



*Fig 5.1* Graph Illustration 1

Has the following parts.

- The underlying set for the Vertices set is the integers.
- The Vertices set =  $\{1, 2, 3, 4, 5, 6\}$
- The Edge set =  $\{(6, 4), (4, 5), (4, 3), (3, 2), (5, 2), (2, 1), (5, 1)\}$

## 2. Kinds of Graphs

Various flavors of graphs have the following specializations and particulars about how they are usually drawn.

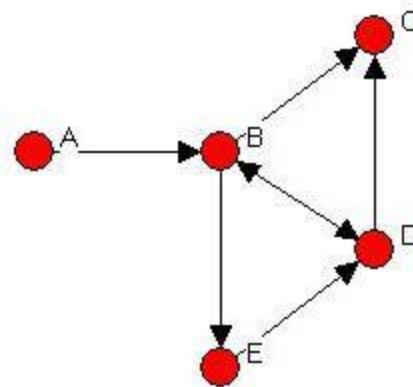
- **Undirected Graphs.**

In an undirected graph, the order of the vertices in the pairs in the Edge set doesn't matter. Thus, if we view the sample graph above we could have written the Edge set as  $\{(4,6),(4,5),(3,4),(3,2),(2,5),(1,2),(1,5)\}$ . Undirected graphs usually are drawn with straight lines between the vertices.

The adjacency relation is symmetric in an undirected graph, so if  $u \sim v$  then it is also the case that  $v \sim u$ .

- **Directed Graphs.**

In a directed graph the order of the vertices in the pairs in the edge set matters. Thus  $u$  is adjacent to  $v$  only if the pair  $(u,v)$  is in the Edge set. For directed graphs we usually use arrows for the arcs between vertices. An arrow from  $u$  to  $v$  is drawn only if  $(u,v)$  is in the Edge set. The directed graph below



*Fig 5.2 Graph Illustration 2*

Has the following parts.

- The underlying set for the Vertices set is capital letters.
- The Vertices set = {A,B,C,D,E}
- The Edge set = {(A,B),(B,C),(D,C),(B,D),(D,B),(E,D),(B,E)}

Note that both (B,D) and (D,B) are in the Edge set, so the arc between B and D is an arrow in both directions.

- Vertex labeled Graphs.

- In a labeled graph, each vertex is labeled with some data in addition to the data that identifies the vertex. Only the identifying data is present in the pair in the Edge set. This is similar to the (key,satellite) data distinction for sorting.

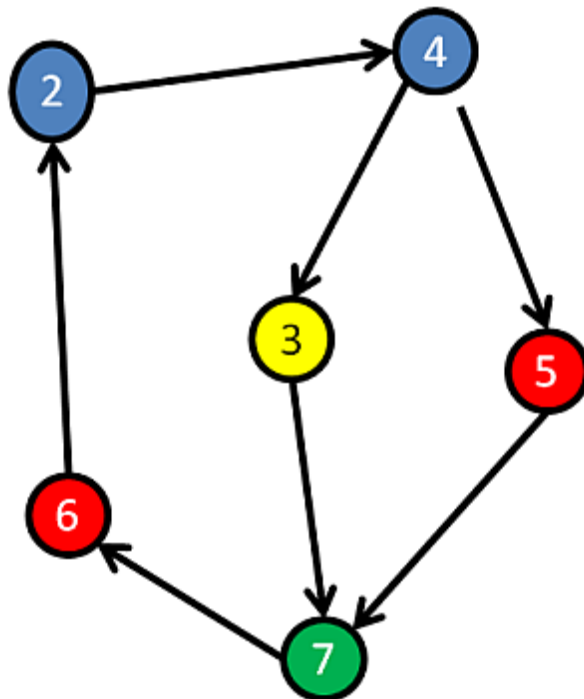


Fig 5.3 Graph Illustration 3

- Here we have the following parts.
- The underlying set for the keys of the Vertices set is the integers.

- The underlying set for the satellite data is Color.
- The Vertices set =  $\{(2, \text{Blue}), (4, \text{Blue}), (5, \text{Red}), (7, \text{Green}), (6, \text{Red}), (3, \text{Yellow})\}$
- The Edge set =  $\{(2,4), (4,5), (5,7), (7,6), (6,2), (4,3), (3,7)\}$

- **Cyclic Graphs.**

- A cyclic graph is a directed graph with at least one cycle. A cycle is a path along the directed edges from a vertex to itself. The vertex labeled graph above as several cycles. One of them is  $2 \gg 4 \gg 5 \gg 7 \gg 6 \gg 2$

- **Edge labeled Graphs.**

A Edge labeled graph is a graph where the edges are associated with labels. One can indicate this by making the Edge set be a set of triples. Thus if  $(u,v,X)$  is in the edge set, then there is an edge from  $u$  to  $v$  with label  $X$

Edge labeled graphs are usually drawn with the labels drawn adjacent to the arcs specifying the edges.

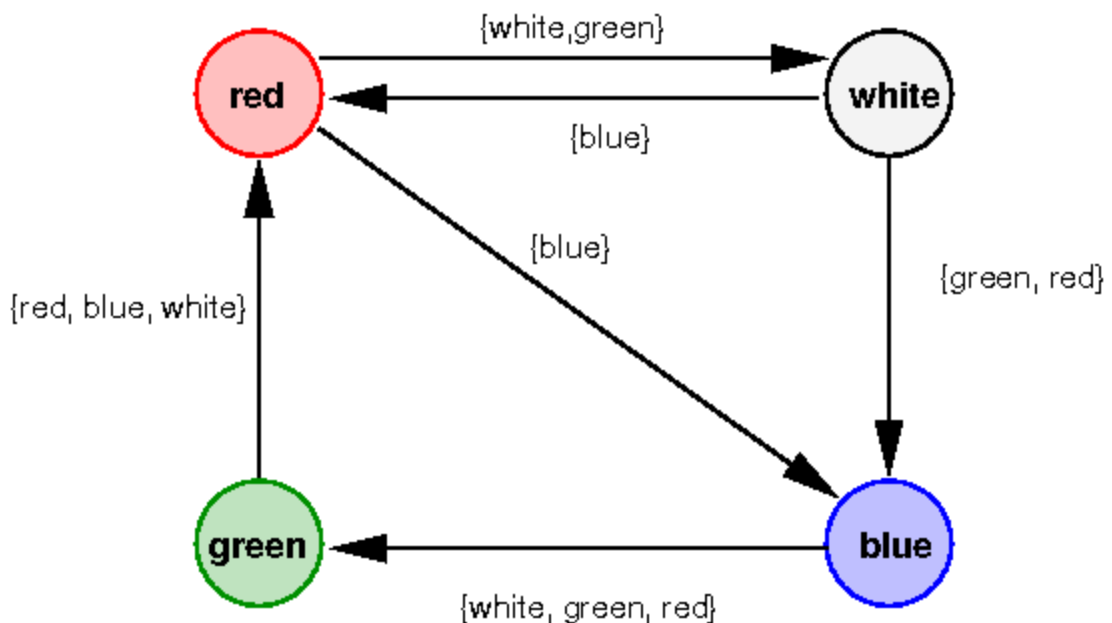


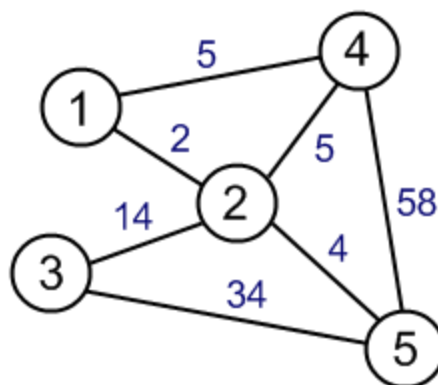
Fig 5.4 Graph Illustration 4

Here we have the following parts.

- The underlying set for the Vertices set is Color.
- The underlying set for the edge labels is sets of Color.
- The Vertices set = {Red,Green,Blue,White}
- The Edge set = {(red,white,{white,green}) ,(white,red,{blue}) , (white,blue,{green,red}) , (red,blue,{blue}) , (green,red,{red,blue,white}) , (blue,green,{ white,green,red})}

- **Weighted Graphs.**

A weighted graph is an edge labeled graph where the labels can be operated on by the usual arithmetic operators, including comparisons like using less than and greater than. In Haskell we'd say the edge labels are in the Num class. Usually they are integers or floats. The idea is that some edges may be more (or less) expensive, and this cost is represented by the edge labels or weight. In the graph below, which is an undirected graph, the weights are drawn adjacent to the edges and appear in dark purple.



*Fig 5.5 Graph Illustration 5*

Here we have the following parts.

- The underlying set for the Vertices set is Integer.
- The underlying set for the weights is Integer.

- The Vertices set = {1,2,3,4,5}
- The Edge set = {(1,4,5) ,(4,5,58) ,(3,5,34) ,(2,4,5) ,(2,5,4) ,(3,2,14) ,(1,2,2)}

- Directed Acyclic Graphs.

- A Dag is a directed graph without cycles. They appear as special cases in CS applications all the time.

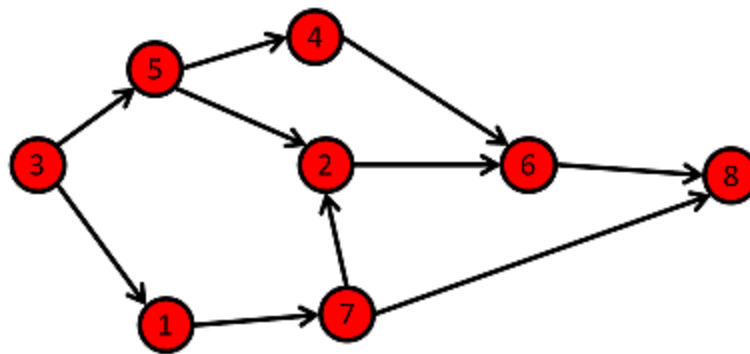


Fig 5.6 Graph Illustration 6

- Here we have the following parts.
- The underlying set for the the Vertices set is Integer.
- The Vertices set = {1,2,3,4,5,6,7,8}
- The Edge set = {(1,7) ,(2,6) ,(3,1),(3,5) ,(4,6) ,(5,4),(5,2) ,(6,8) ,(7,2),(7,8)}

- Disconnected Graphs

- Vertices in a graph do not need to be connected to other vertices. It is legal for a graph to have disconnected components, and even lone vertices without a single connection.
- Vertices (like 5,7,and 8) with only in-arrows are called sinks. Vertices with only out-arrows (like 3 and 4) are called sources.
- Here we have the following parts.



- The underlying set for the the Vertices set is Integer.
- The Vertices set = {1,2,3,4,5,6,7,8}
- The Edge set = {(1,7) ,(3,1),(3,8) ,(4,6) ,(6,5)}

### 3. Representing graphs in a computer

Graphs are often used to represent physical entities (a network of roads, the relationship between people, etc) inside a computer. There are numerous mechanisms used. A good choice of mechanism depends upon the operations that the computer program needs to perform on the graph to achieve its needs. Possible operations include.

Compute a list of all vertices

Compute a list of all edges.

For each vertex,  $u$ , compute a list of edges  $(u,v)$ . This is often called the adjacency function.

If the graph is labeled (either vertex labeled or edge labeled) compute the label for each vertex (or edge).

Not all programs will need all of these operations, so for some programs, an efficient representation that can compute only the operations needed (but not the others), will suffice.

- **Graphs as sets.**

One way to represent graphs would be to directly store the Vertices set and the Edge set. This can make it difficult to efficiently compute adjacency information for particular vertexes quickly, so this representation is not used too often.

- **Graphs as adjacency information.**

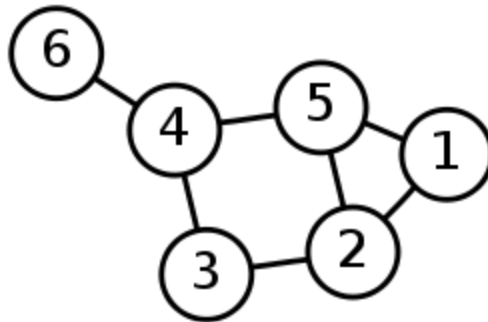
Most programs need to compute all the vertices adjacent to a given vertex. This corresponds to finding a 1-step path in the graph. In fact, for many programs this is the only operation needed, so data structures that support this operation quickly and efficiently are often used. Possible choices include arrays, balanced trees, hash tables, etc.

- Graphs as functions.

One useful abstraction is to think of the adjacency information as a function. Under this abstraction a graph is nothing more than a function.

```
type Graph vertex = vertex -> [vertex]
```

For example the undirected graph below:



*Fig 5.7 Graph Illustration 7*

can be represented as the function.

```
graph1:: Graph Int
```

```
graph1 6 = [4]
```

graph1 5 = [1,2,4]

graph1 4 = [3,5,6]

graph1 3 = [4,2]

graph1 2 = [1,3,5]

graph1 1 = [2,5]

graph1 \_ = [ ]

This mechanism can be extended to a wide variety of graphs types by slightly altering or enhancing the kind of function that represents the graph. Here are a few examples.

- Directed graph.

type Dgraph vertex = vertex -> [vertex]

The representation is the same as a undirected graph but the interpretation is different. In

an undirected graph, f, with edge (2,3), we would have both

f 2 ---> [3, ... ]

f 3 ---> [2, ... ]

but in a directed graph we would have only the first of the results. Consider the directed graph below:

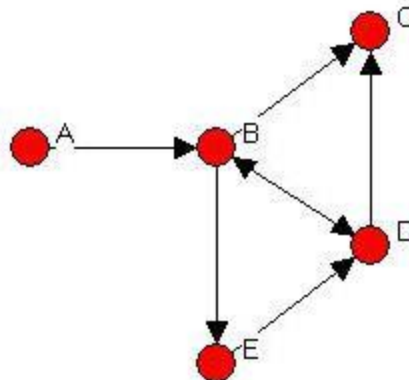


Fig 5.8 Graph Illustration 8

We could represent this as a Dgraph as follows:

```
data Node = A | B | C | D | E
```

```
graph2:: Dgraph Node
```

```
graph2 A = [B]
```

```
graph2 B = [C,D,E]
```

```
graph2 C = []
```

```
graph2 D = [B,C]
```

```
graph2 E = [D]
```

```
graph2 _ = []
```

- Vertex labeled graph.

```
type VLgraph label vertex = vertex -> ([vertex],label)
```

Here the function not only returns the adjacency list for a vertex but also the label. For example:

```
data Color = Blue | Red | Yellow | Green
```

```
graph4:: VLgraph Color Int
```

```
graph4 2 = ([4],Blue)
```

```
graph4 3 = ([7],Yellow)
```

```
graph4 4 = ([3,5],Blue)
```

```
graph4 5 = ([7],Red)
```

```
graph4 6 = ([2],Red)
```

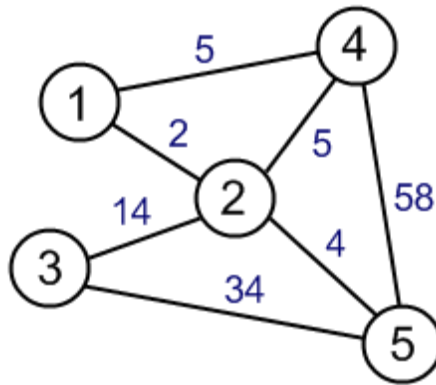
```
graph4 7 = ([6],Green)
```

```
graph4 _ = ([],undefined)
```

- Edge labeled graph.

```
type ELgraph label vertex = vertex -> [(vertex,label)]
```

Here, the adjacency list now contains a tuple, the adjacent vertex, and the label of edge to that vertex



*Fig 5.9 Graph Illustration 9*

```
graph6:: ELgraph Int Int
```

```
graph6 1 = [(4,5),(2,2)]
```

```
graph6 2 = [(1,2),(4,5),(3,14),(5,4)]
```

```
graph6 3 = [(2,14),(5,34)]
```

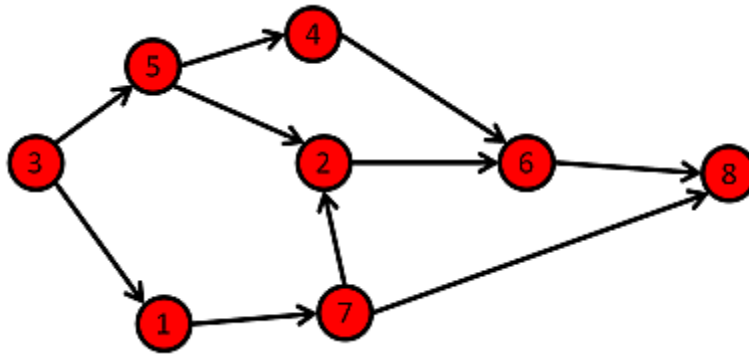
```
graph6 4 = [(1,5),(2,5),(5,58)]
```

```
graph6 5 = [(2,4),(3,34),(4,58)]
```

```
graph6 _ = []
```

- DAG.

Here we have a simple graph, but the data must meet some invariants ensuring no cycles



*Fig 5.10* Graph Illustration 10

graph7:: Graph Int

graph7 1 = [7]

graph7 2 = [6]

graph7 3 = [1,5]

graph7 4 = [6]

graph7 5 = [2,4]

graph7 6 = [8]

graph7 7 = [2,8]

graph7 8 = []

graph7 \_ = []

- Advantages of representing graphs as functions
  1. Simple and easy to understand
  2. Adapts easily to different kinds of graphs
- Disadvantages of using graphs as functions
  1. Cannot be extended to accommodate queries about the set of Vertices or the set of Edges.
  2. Depending upon the compiler that compiles the functions may not be very efficient. In fact the worst case time could be proportional to the number of vertices.
  3. The graph must be known statically at compile time.
- Graphs as arrays of adjacent vertexes.

One mechanism that can ameliorate the disadvantages of using functions as a way to represent graphs is to use arrays instead. Using this mechanism requires that the underlying domain of Vertices be some type that can be used as indexes into an array.

In the rest of this note we will assume that Vertices are of type `Int`, and that the Vertices set is a finite range of the type `Int`. Thus a graph can be represented as follows:

```
type ArrGraph = Array [Int]
```

We can now answer a number queries about graphs quickly and efficiently.

```
type ArrGraph i = Array [i]
```

```
vertices:: ArrGraph i -> IO[Int]
```

```
edges:: ArrGraph i -> IO[(Int,i)]
```

```
children:: ArrGraph i -> i -> IO[i]
```

```
vertices g =
```

```
do { (lo,hi) <- boundsArr g
```

```
; return [lo..hi]}
```

```
edges g =
```

```
do { (lo,hi) <- boundsArr g
```

```
; ees <- toListArr g
```

```
; return [ (i,j) | (i,cs) <- zip [lo..hi] ees, j <- cs ] }
```

```
children g node = readArr g node
```

- Advantages of representing graphs as arrays
  1. Simple and easy to understand
  2. Efficient access
  3. Graphs can be constructed at run-time
  4. Adapts easily to different kinds of graphs
  5. type VLArrGraph label = Array ([Int],label) -- Vertex labeled graphs
  6. type ELArrGraph label = Array [(Int,label)] -- Edge labeled graphs
- Disadvantages of representing graphs as arrays
  1. Requires that graph access be a Command rather than a computation.
  2. The domain of Vertices must be a type that can be used as an index into an array.



# CHAPTER 6

## GRAPH COMPRESSION

### 1. Introduction

I propose to compress weighted graphs (networks), motivated by the observation that large networks of social, biological, or other relations can be complex to handle and visualize. In the process also known as graph simplification, nodes and (unweighted) edges are grouped to supernodes and superedges, respectively, to obtain a smaller graph. I propose a model and algorithm for weighted graphs. The interpretation (i.e. decompression) of a compressed, weighted graph is that a pair of original nodes is connected by an edge if their supernodes are connected by one, and that the weight of an edge is approximated to be the weight of the superedge. The compression problem now consists of choosing supernodes, superedges, and superedge weights so that the approximation error is minimized while the amount of compression is maximized.

Here, I formulate this task as the 'simple weighted graph compression problem'. I then propose a much wider class of tasks under the name of 'generalized weighted graph compression problem'. The generalized task extends the optimization to preserve longer-range connectivities between nodes, not just individual edge weights. I study the properties of these problems and propose an algorithm to solve them, with different balances between complexity and quality of the result. I evaluate the problems and algorithms experimentally on real and dummy networks. The results indicate that weighted graphs can be compressed efficiently with little error.

Graphs and networks are used in numerous applications to describe relationships between entities, such as social relations between persons, links between web pages, flow of traffic, or interactions between proteins. In many applications, relationships have weights that are central to any use or analysis of graphs: how frequently do two persons communicate or how much do they influence each other's opinions; how much web traffic flows from one page to another or how many cars drive from one crossing to another; or how strongly does one protein regulate the other one?

Here, I discuss models and methods for the compression of weighted graphs into smaller graphs that contain approximately the same information. In this process, also known as graph simplification in the context of unweighted graphs, nodes are grouped to supernodes, and edges are grouped to superedges between supernodes. A superedge then represents all possible edges between the pairs of nodes in the adjacent supernodes.

This problem is different from graph clustering or partitioning where the aim is to find groups of strongly related nodes. In graph compression, nodes are grouped based on the similarity of their relationships to other nodes, not by their (direct) mutual relations.

As a small example, consider the co-authorship social network in Figure 6.1a. It contains an excerpt from the DBLP Computer Science Bibliography 1, a subgraph containing Jiawei Han and Philip S. Yu and a dozen related authors. Nodes in this graph represent authors and edges represent co-authorships. Edges are weighted by the number of coauthored articles. Compressing this graph just by about 30% gives a simpler graph that highlights some of the inherent structure or roles in the original graph (Figure 6.1b).

For instance, Ke Wang and Jianyong Wang have identical sets of co-authors (in this excerpt from DBLP) and have been grouped together. This is also an example of a group that would not be found by traditional graph clustering methods, since the two nodes grouped together are not directly connected. Daxin Jiang and Aidong Zhang have been grouped, but additionally the self-edge of their supernode indicates that they have also authored papers together. Groups that could not be obtained by the existing compression algorithms of can be observed among the six authors that (in this excerpt) only connect to Jiawei Han and Philip S. Yu.

Instead of being all grouped together as structurally equivalent nodes, we have three groups that have different weight profiles. Charu C. Aggarwal is a group by himself, very strongly connected with Philip S. Yu. A second group includes Jiong Yang, Wei Fan, and Xifeng Yan, who are roughly equally strongly connected to both Jiawei Han and Philip S. Yu. The third group, Hong Cheng and Xiaoxin Yin, are more strongly connected to Jiawei Han. Such groups are not found with methods for unweighted graphs.

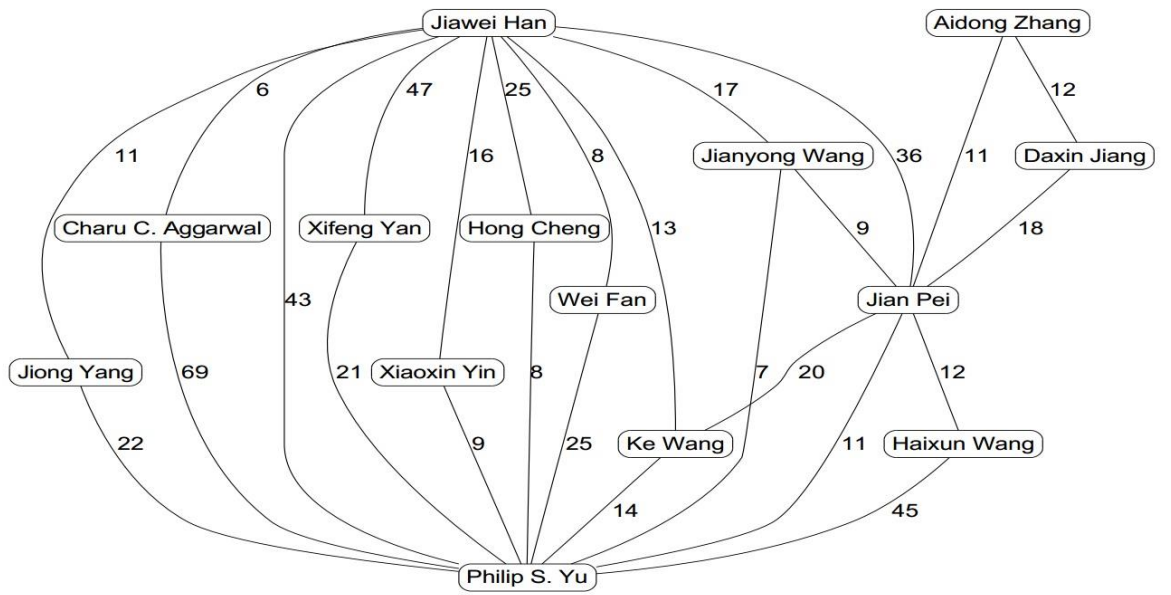


Fig 6.1 Uncompressed Graph

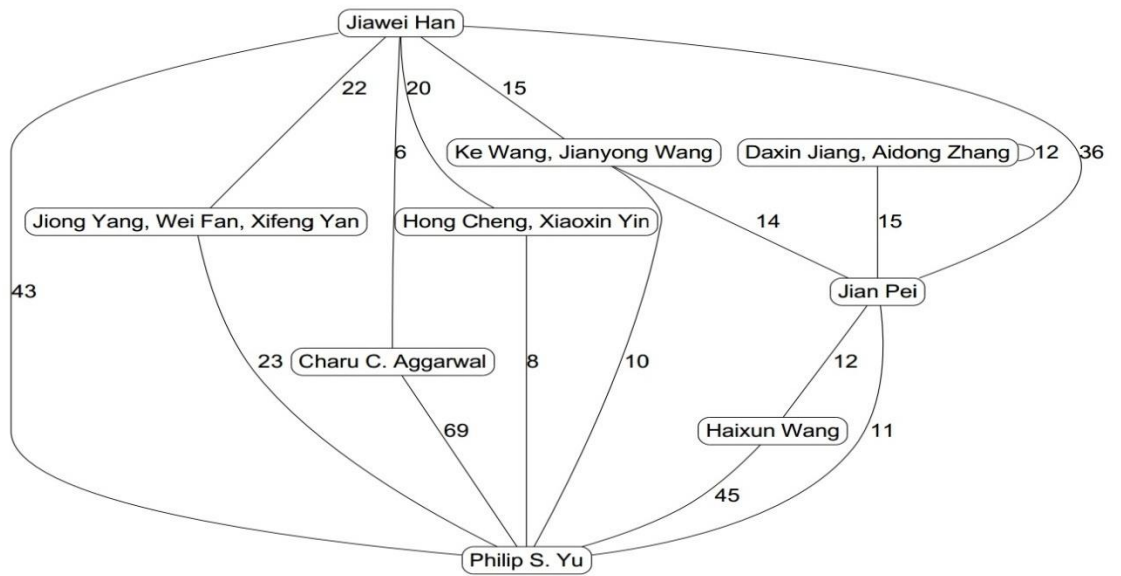


Fig 6.1 Compressed Graph

In what we define as the simple weighted graph compression problem, the approximation error of the compressed graph with respect to original edge weights is minimized by assigning each superedge the mean weight of all edges it represents. For many applications on weighted graphs it is, however, important to preserve relationships between faraway nodes, too, not just individual edge weights. Motivated by this, I also introduce the generalized weighted graph compression problem where the goal is to produce a compressed graph that maintains connectivities across the graph: the best path between any two nodes should be approximately equally good in the compressed graph as it is in the original graph, but the path does not have to be the same.

Compressed weighted graphs can be utilized in a number of ways. Graph algorithms can run more efficiently on a compressed graph, either by considering just the smaller graph consisting of supernodes and superedges, or by decompressing parts of it on the fly when needed. An interesting possibility is to provide an interactive visualization of a graph where the user can adjust the abstraction level of the graph on the fly.

## 2. Simple Weighted Graph Compression

Compression ratio does not consider the amount of errors introduced in edges and their weights. This issue is addressed by a measure of dissimilarity between graphs. We first present a simple distance measure that leads to the simple weighted graph compression problem.

**Definition :** The simple distance between two graphs  $G_a = (V, E_a, w_a)$  and  $G_b = (V, E_b, w_b)$ , with an identical set of nodes  $V$ , is

$$dist_1(G_a, G_b) = \sqrt{\sum_{\{u,v\} \in V \times V} (w_a(\{u,v\}) - w_b(\{u,v\}))^2}$$

This distance measure has an interpretation as the Euclidean distance between  $G_a$  and  $G_b$  in a space where each pair of nodes  $\{u, v\} \in V \times V$  has its own dimension. Given the distance definition, the dissimilarity between a graph  $G$  and its compressed representation  $S$  can then be defined simply as  $\text{dist1}(G, \text{dec}(S))$ . The distance can be seen as the cost of compression, whereas the compression ratio represents the savings. Our goal is to produce a compressed graph which optimizes the balance between these two. In particular, we will consider the following form of the problem. Definition Given a weighted graph  $G$  and a compression ratio  $cr$ ,  $0 < cr < 1$ , the simple weighted graph compression problem is to produce a compressed representation  $S$  of  $G$  with  $cr(S) \leq cr$  such that  $\text{dist1}(G, \text{dec}(S))$  is minimized.

Other forms can be just as useful. One obvious choice would be to give a maximum distance as parameter, and then seek for a minimum compression ratio. In either case, the problem is complex, as the search space consists of all partitions of  $V$ . However, the compression ratio is non-increasing and graph distance non-decreasing when nodes are merged to supernodes, and this observation can be used to devise heuristic algorithms for the problem.

Given a compressed graph structure, it is easy to set the weights of superedges to optimize the simple distance measure  $\text{dist1}(\cdot)$ . Each pair  $\{u, v\} \in V \times V$  of original nodes is represented by exactly one pair  $\{u_0, v_0\} \in V_0 \times V_0$  of supernodes, including the cases  $u = v$  and  $u_0 = v_0$ . In order to minimize Equation 1, given the supernodes  $V_0$ , we need to minimize for each pair  $\{u_0, v_0\}$  of supernodes the sum  $\sum_{\{u, v\} \in u_0 \times v_0} (w(\{u, v\}) - w_0(\{u_0, v_0\}))^2$ . This sum is minimized when the superedge weight is the mean of the original edge weights (including “zero-weight edges” for those pairs of nodes that are not connected by an edge):

$$w'(\{u', v'\}) = \frac{\sum_{\{u, v\} \in u' \times v'} w(\{u, v\})}{|u'| |v'|}$$

where  $|x|$  is the number of original nodes in supernode  $x$ . The compression algorithms that we propose below work in an incremental, often greedy fashion, merging two supernodes at a time into a new supernode (following the ideas of references [12, 14]). The merge operation that these algorithms use is specified in Algorithm 1. It takes a graph and its two nodes as parameters, and it returns a graph where the given nodes are merged into one and the edge weights of the new supernode are set according to Equation 3. Line 6 of the merge operation sets the weight of the self-edge for the supernode. When  $\lambda = 1$ , function  $W(x,y)$  returns the sum of weights of all original edges between  $x$  and  $y$  using their mean weight  $Q1(\{x,y\}; S)$ . The weight of the self-edge is then zero and the edge non-existent if neither  $u$  or  $v$  has a self-edge and if there is no edge between  $u$  and  $v$ .

Setting superedge weights optimally is much more complicated for the generalized distance (Equation 2) when  $\lambda > 1$ : edge weights contribute to best paths and therefore distances up to  $\lambda$  hops away, so the distance cannot be optimized in general by setting each superedge weight independently. I use the merge operation of Algorithm 1 as an efficient, approximate solution also in these cases, and leave more optimal solutions for future work

### 3. Merge Algorithm

Algorithm 1 :  $\text{merge}(u,v,S)$

Input: Nodes  $u$  and  $v$ , and a compressed graph  $S =$

$(V,E,w)$  s.t.  $u,v \in V$

Output: A compressed graph  $S_0$  obtained by merging  $u$

and  $v$  in  $S$

1:  $S_0 \leftarrow S$  {i.e.,  $(V_0,E_0,w_0) \leftarrow (V,E,w)$ }

2:  $z \leftarrow \{u \cup v\}$

3:  $V_0 \leftarrow V_0 \setminus \{u,v\} \cup \{z\}$

```

4: for all  $x \in V$  s.t.  $u \neq x \neq v$ , and  $\{u,x\}$  or  $\{v,x\} \in E$ 
do
5:  $w_0(\{z,x\}) = |u| Q_\lambda(\{u,x\}; S_u) + |v| Q_\lambda(\{v,x\}; S)$ 
6:  $w_0(\{z,z\}) = W(u,u) + zW(|z|-v,v) + 2W(u,v)$ 
7: return  $S_0$ 
8: function  $W(x,y)$ :
9: if  $x \neq y$  then
10: return  $Q_\lambda(\{x,y\}; S) |x||y|$ 
11: else
12: return  $Q_\lambda(\{x,x\}; S) |x|(|x| - 1)/2$ 

```

## 4. Compression Algorithm

This algorithm has the following input and output:

**Input:** weighted graph  $G = (V, E, w)$ , compression ratio  $cr$  ( $0 < cr < 1$ ), path quality function  $q$ , and maximum path length  $\lambda \in \mathbb{N}$ .

**Output:** compressed weighted graph  $S = (V_0, E_0, w_0)$  with  $cr(S) \leq cr$ , such that  $\text{dist}(G, \text{dec}(S))$  is minimized.

**Brute-force greedy algorithm.** The brute-force greedy method computes the effects of all possible pairwise mergers (Line 4) and then performs the best merger (Line 5), and repeats this until the requested compression rate is achieved. The algorithm generalizes the greedy algorithm of Navlakha et al. to distance functions  $\text{dist}_\lambda(\cdot)$  that take the maximum path length  $\lambda$  and the path quality function  $q$  as parameters.

Brute-force greedy search

```
1:  $S \leftarrow G$  {i.e.,  $(V_0, E_0, w_0) \leftarrow (V, E, w)$ }  
2: while  $cr(S) > cr$  do  
3: for all pairs  $\{u, v\} \in V_0 \times V_0$  do  $\{(*)\}$   
4: d  
 $\{u,v\} \leftarrow \text{dist}(G, \text{dec}(\text{merge}(u, v, S)))$   
5:  $S \leftarrow \text{merge}(\arg \min_{\{u,v\}} d_{\{u,v\}}, S)$   
6: return  $S$ 
```

(\*) 2-hop optimization can be used, see text.

The worst-case time complexity for simple weighted graph compression is  $O(|V|^4)$ , and for generalized compression  $O(|V|^3|E| \log |V|)$ . I omit the details for brevity.

**2-hop optimization** : The brute-force method, as well as all other methods we present here, can be improved by the 2-hop optimization. Instead of arbitrary pairs of nodes, the 2-hop optimized version only considers  $u$  and  $v$  for a potential merger if they are exactly two hops from each other. Since 2-hop neighbors have a shared neighbor that can be linked to the merged supernode with a single superedge, some compression may result. The 2-hop optimization is safe in the sense that any merger by Algorithm 1 that compresses the graph involves 2-hop neighbors. The time saving by 2-hop optimization can be significant:

for the brute-force method, for instance, there are approximately  $O(\text{deg} |E|)$  feasible node pairs with the optimization, where  $\text{deg}$  is the average degree, instead of the  $O(|V|^2)$  pairs in the unoptimized algorithm. For the randomized methods below, a straight-forward implementation of 2-hop optimization by random walk has a nice property. Assume that one node has been chosen, then find a random pair for it by taking two consecutive



random hops starting from the first node. Now 2-hop neighbors with many shared neighbors are more likely to get picked, since there are several 2-hop paths to them. Such pairs, with many shared neighbors, lead to better compression. A uniform selection among all 2-hop neighbors does not have this property.

## Conclusion and Future Work

Data compression is a topic of much importance and many applications. Methods of data compression have been studied for almost four decades. This report has provided an overview of data compression methods of general utility. The algorithms have been evaluated in terms of the amount of compression they provide, algorithm efficiency, and susceptibility to error. While algorithm efficiency and susceptibility to error are relatively independent of the characteristics of the source ensemble, the amount of compression achieved depends upon the characteristics of the source to a great extent.

Semantic dependent data compression techniques, are special-purpose methods designed to exploit local redundancy or context information. A semantic dependent scheme can usually be viewed as a special case of one or more general-purpose algorithms. It should also be noted that algorithm BSTW is a general-purpose technique which exploits locality of reference, a type of local redundancy.

Susceptibility to error is the main drawback of each of the algorithms presented here. Although channel errors are more devastating to adaptive algorithms than to static ones, it is possible for an error to propagate without limit even in the static case. Methods of limiting the effect of an error on the effectiveness of a data compression algorithm should be investigated.

Implementation of Adaptive Huffman to compress/encode text files is provided in the CD along with data set on which the program is used.

Since Graph Compression is a vast area, I intend to pursue learning and implementing other forms of Graph Compression algorithms.

## References

- [1] Lelewer and Hirschberg, <http://www.ics.uci.edu/~dan/pubs/DC-Sec1.html>
- [2] Dave Marshall, <http://www.cs.cf.ac.uk/Dave/Multimedia/node212.html>
- [3] Abramson, N. 1963. *Information Theory and Coding*. McGraw-Hill, New York.
- [4] Connell, J. B. 1973. A Huffman-Shannon-Fano Code. *Proc. IEEE* 61, 7 (July), 1046-1047.
- [5] Cortesi, D. 1982. An Effective Text-Compression Algorithm. *BYTE* 7, 1 (Jan.), 397-403.
- [6] Knuth, D. E. 1971. Optimum Binary Search Trees. *Acta Inf.* 1, 1 (Jan.), 14-25.
- [7] Knuth, D. E. 1985. Dynamic Huffman Coding. *J. Algorithms* 6, 2 (June), 163-180.
- [8] Vitter, J. S. 1987. Design and Analysis of Dynamic Huffman Codes. *J. ACM* 34, 4 (Oct.), 825-845.
- [9] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In Data Compression Conference, pages 203–212, 2001
- [10] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In WWW '04: Proceedings of the 13th international conference on World Wide Web, pages 595–602, New York, NY, USA, 2004. ACM.
- [11] S. P. Borgatti and M. G. Everett. Regular blockmodels of multiway, multimode matrices. *Social Networks*, 14:91–120, 1992.
- [12] C. Chen, C. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. In 2009 Int. Conf. on Very Large Data Bases, pages 742–753, Lyon, France, August 2009. VLDB Endowment.
- [13] C. Chen, X. Yan, F. Zhu, J. Han, and P. Yu. Graph OLAP: Towards online analytical processing on graphs. In ICDM '08: Proceedings of the 2008 Eighth IEEE

International Conference on Data Mining, pages 103–112, Washington, DC, USA, 2008. IEEE Computer Society

[14] U. Elsner. Graph partitioning - a survey. Technical Report SFB393/97-27, Technische Universität at Chemnitz, 1997.

[15] C. Faloutsos, K. S. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 118–127, New York, NY, USA, 2004. ACM.

[16] P.-O. Fjällström. Algorithms for graph partitioning: A Survey. In Linköping Electronic Articles in Computer and Information Science, 3, 1998.

[17] S. Hauguel, C. Zhai, and J. Han. Parallel PathFinder Algorithms for Mining Structures from Graphs. In 2009 Ninth IEEE International Conference on Data Mining, pages 812–817. IEEE, 2009.

[18] P. Hintsanen and H. Toivonen. Finding reliable subgraphs from large probabilistic graphs. *Data Mining and Knowledge Discovery*, 17:3–23, 2008.

[19] F. Lorrain and H. C. White. Structural equivalence of individuals in social networks. *Journal of Mathematical Sociology*, 1:49–80, 1971.

[20] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 419–432, New York, NY, USA, 2008. ACM.

[21] S. Navlakha, M. Schatz, and C. Kingsford. Revealing Biological Modules via Graph Summarization. Presented at the RECOMB Systems Biology Satellite Conference. *J. Comp. Bio.*, 16:253–264, 2009.

[22] Y. Tian, R. Hankins, and J. Patel. Efficient aggregation for graph summarization. In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 567–580, New York, NY, USA, 2008. ACM.

- [23] H. Toivonen, S. Mahler, and F. Zhou. A framework for path-oriented network simplification. In *Advances in Intelligent Data Analysis IX*, volume 6065/2010, pages 220–231, Berlin/Heidelberg, May 2010. Springer-Verlag.
- [24] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.
- [25] N. Zhang, Y. Tian, and J. Patel. Discovery-driven graph summarization. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 880–891. IEEE, 2010.