Techniques and Applications of Data Compression in Information Retrieval Systems

Shmuel T. Klein
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan 52900, Israel
tomi@cs.biu.ac.il
http://www.biu.ac.il/~tomi

April 18, 2000

Tel number: ++(972–3) 531 8865

Fax: ++(972–3) 535 3325

Contents

List of Tables and Figures

Techniques and Applications of Data Compression in Information Retrieval Systems

1. INTRODUCTION

As can be seen from the title, we shall concentrate on techniques that are at the crossroad of two disciplines: Data Compression (DC) and Information Retrieval (IR). Each of these encompass a large body of knowledge that has evolved over the last decades, each with its own philosophy and its own scientific community. Nevertheless, their intersection is particularly interesting, the various files of large full-text IR systems providing a natural testbed for new compression methods, and DC enabling the proliferation of improved retrieval algorithms.

A chapter about data compression in a book published at the beginning of the twenty first century might at a first glance seem anachronistic. Critics will say that storage space is getting cheaper every day, tomorrow it will be almost given for free, so who needs complicated methods to save a few bytes.... What these critics overlook, is that for data storage, supply drives demand: our appetite for getting ever increasing amounts of data into electronic storage grows just as steadily as does the standard size of the hard disk in our current personal computer. Most users know that whatever the size of their disks, they will fill up sooner or later, and generally sooner than they wish.

However, there are also other benefits to be gained from data compression, beyond the reduction of storage space. One of the bottlenecks of our computing systems is still the slow data transfer from external storage devices. Similarly, for communication applications, the problem is not to store the data but rather to squeeze it through some channel. But many users are competing for the same limited bandwidth, effectively reducing the amount of data that can be transferred in a given time span. Here, DC may help reduce the number of I/O operations to and from secondary memory, and for communication it reduces the actual amount of data that has to pass through the channel. The additional time spent on compression and decompression is generally largely compensated for by the savings in transfer time.

For these reasons, research in DC is not dying out, but just the opposite is true, as evidenced by the recent spurt of literature in this area. An international Data Compression Conference convenes annually since 1990, and many journals, including even popular ones such as Byte, Dr. Dobbs, IEEE Spectrum, Datamation, PC Magazine and others, have repeatedly published articles on compression recently.

It is true that a large part of the research concentrates on image compression. Indeed, pictorial data is storage voracious so that the expected profit of efficient compression is substantial. The techniques generally applied to images belong to the class of *lossy* compression, because they concentrate on how to throw away part of the data, without too much changing its general appearance. For instance, most humans do not really see any difference between a picture coded with 24 bits per pixel, allowing more than 16 million colors, and the same picture recoded with 12 bits per pixel, giving "only" about 4000 different color shades. Of course, most image compression techniques are much more sophisticated, but we shall not deal with them in the present survey. The interested reader is referred to the large literature on lossy compression, e.g. [1].

Information Retrieval is concerned, on the one hand, with procedures to help a user satisfy his information needs by facilitating his access to large amounts of data, on the other hand, with techniques to evaluate his (dis)satisfaction with whatever data the system provided. We shall concentrate primarily on the algorithmic aspects of IR. A functional full-text retrieval system is constituted of a large variety of files, most of which can and should be compressed. Some of the methods described below are of general applicability, and some are specially designed for an IR environment.

Full-text information retrieval systems may be partitioned according to the level of specificity supported by their queries. For example, in a system operating at the *document*-level, queries can be formulated as to the presence of certain keywords in each document of the database, but not as to their exact locations within the document. Similarly, one can define the *paragraph*-level and *sentence*-level, each of which is a refinement of its predecessor. The highest specificity level is the *word*-level, in which the requirement is that the keywords appear within specified distances of each other. With such a specificity level, one could retrieve all the occurrences of A and B such that there are at least two but at most five words between them. In the same way, the paragraph and sentence-levels permit also appropriate distance constraints, e.g., at the sentence-level one could ask for all the occurrences of A and B in the same or adjacent sentences.

Formally, a typical query consists of an optional level-indicator, m keywords and $m-1$ distance constraints, as in

$$\textit{level}:\ A_1\,(l_1,u_1)\,A_2\,(l_2,u_2)\ \cdots\ A_{m-1}\,(l_{m-1},u_{m-1})\,A_m. \qquad (1.1)$$

The l_i and u_i are (positive or negative) integers satisfying $l_i \le u_i$ for $1 \le i < m$, with the couple (l_i, u_i) imposing a lower and upper limit on the distance from A_i to A_{i+1}. Negative distance means that A_{i+1} may appear before A_i in the text. The distance is measured in words, sentences or paragraphs, as prescribed by the level-indicator. In case the latter is omitted, word-level is assumed; in this case, constraints of the form $A\,(1,1)\,B$ (meaning that A should be followed immediately by B), are omitted. Also, if the query is on the document level, then the distances are meaningless and should be omitted (the query degenerates then into a conjunction of the occurrences of all the keywords in the query).

In its simplest form, the keyword A_i is a single word or a (usually very small) set of words given explicitly by the user. In more complex cases a keyword A_i in (1.1) will represent a set of words $A_i = \bigcup_{j=1}^{n_i} A_{ij}$, all of which are considered synonymous to A_i in the context of the given query. For example, a variable-length-don't-care-character $*$ can be used, which stands for an arbitrary, possibly empty, string. This allows the use of prefix, suffix and infix truncation in the query. Thus A_i could be `comput*`, representing, among others, the words `computer`, `computing`, `computerize`, etc.; or it could be `*mycin`, which retrieves a large class of antibiotics; infix truncation also can be useful for spelling foreign names, such as `Ba*tyar`, where $*$ could be matched by `h`, `k`, `kh`, `ch`, `sh`, `sch`, etc.

Another possibility for getting the variants of a keyword is from the use of a *thesaurus* (`month` representing `January`, `February`, etc.), or from some morphological processing (`do` representing `does`, `did`, `done`, etc.). Although these grammatical variants can be easily generated in some languages with simple morphology like English, sophisticated linguistic tools are needed for languages such as Hebrew, Arabic and many others. One of the derivatives of the 2-character word `daughter` in Hebrew, for example, is a 10-character string meaning `and when our daughters`, and it shares only *one* common letter with its original stem; a similar phenomenon occurs in French with the verb `faire`, for example.

For all these cases, the families A_i are constructed in a preprocessing stage. Algorithms for generating the families identified by truncated terms can be found in [2], and for the families of grammatical variants in [3].

This general definition of a query with distance constraints allows great flexibility in the formulation of the query. For example: the query `solving` (1,3) `differential equations` will retrieve sentences containing `solving differential equations`, as well as `solving these differential equations` and `solving the required differential equations`, but not `solving these systems of differential equations`. The query `true` (-2,2) `false` can be used to retrieve the phrases `true or false` and `false or true`; since these words appear frequently in some mathematical texts, searching for `true` and `false` in the same sentence could generate noise. A lower bound greater than 1 in the distance operators is needed for example when one wishes to locate phrases in which some words $X_1, X_2, \ldots$ appear, but the juxtaposition of these words $X_1 X_2 \cdots$ forms an idiomatic expression which we do not wish to retrieve. For example, ...`the security of the council members assembled here`... should be retrieved by the query `security` (2,4) `council`. Note however that (1) implies that one can impose distance constraints only on adjacent keywords. In the query `A` (1,5) `B` (2,7) `C`, the pair (2,7) refers to the distance from `B` to `C`. If we wish to impose positive bounds on the distances from `A` to both `B` and `C`, this can be done by using negative distances: `C` (-7,-2) `A` (1,5) `B`, but this procedure cannot be generalized to tying more than two keywords to `A`.

A well-known problem in retrieval systems is the handling of "negative" keywords, i.e., words the *absence* of which, in a specified distance from a specified context, is required. A negation operator (represented here by the minus sign $-$) is particularly useful for excluding known homonyms so as to increase precision. For example, searching for references to the former US President, one could submit the query `Reagan` (-2,1) $-$`Donald`. Another interesting example would be to use the constraints $(l_i, u_i) = (0, 0)$ in order to restrict some large families of keywords, as in the example `comput`$*$ (0,0) $-$`computer`$*$, which would retrieve `computing, computation,` etc, but not `computer` or `computers`. The general definition of a query as given in (1.1) should therefore include the possibility of negating some — but not all — of the keywords while specifying their appropriate distance constraints.

Queries of type (1.1) can be of course further combined by the Boolean operators of AND, OR and NOT, but we shall restrict our attention here to queries of type (1.1), since they are quite common on one hand, and on the other hand their efficient processing is anyway a prerequisite to the efficient processing of the more complicated ones.

At the end of the search process, the solutions are presented to the user in the form of a list of the identifying numbers or the titles of the documents that contain at least one solution, possibly together with the text of the sentence (or the paragraph), in which this solution occurs. The exact details of the display depend on the specific system, on the target population and on the human-interface design of the system.

The way to process such queries depends on the size of the database. When the size of the text is small, say up to a few hundred Kbytes, the problem of efficiently accessing the data can generally be solved by some brute-force method that scans the whole text in reasonable time. Such a method is commonly used in text-editors. At the other extreme, for very large databases spanning hundreds of Mbytes, a complete scan is not feasible. The usual approach in that case is to use so-called *inverted files*.

Every occurrence of every word in the database can be uniquely characterized by a sequence of numbers that give its exact position in the text; typically, in a word-level retrieval system, such a sequence would consist of the document-number, the paragraph number (in the document), the sentence number (in the paragraph), and the word number (in the sentence). These are the *coordinates* of the occurrence. For every word W, let $\mathcal{C}(W)$ be the ordered list of the coordinates of all its occurrences in the text. The problem of processing a query of type (1) consists then, in its most general form, of finding all the m-tuples $(a_1, \ldots, a_m)$ of coordinates satisfying

$$\forall i \in \{1, \ldots, m\} \quad \exists j \in \{1, \ldots, n_i\} \qquad \text{with} \quad a_i \in \mathcal{C}(A_{ij})$$

and

$$l_i \leq d(a_i, a_{i+1}) \leq u_i \qquad \text{for } 1 \leq i < m,$$

where $d(x, y)$ denotes the distance from x to y on the given level. Every m-tuple satisfying these two equations is called a *solution*.

In the inverted files approach, processing (1.1) does not involve directly the original text files, but rather the auxiliary *dictionary* and *concordance* files. The concordance contains, for each distinct word W in the database, the ordered list $\mathcal{C}(W)$ of all its coordinates in the text; it is accessed via the dictionary that contains for every such word a pointer to the corresponding list in the concordance. For each keyword A_i in (1.1) and its attached variants A_{ij}, the lists $\mathcal{C}(A_{ij})$ are fetched from the concordance and merged to form the combined list $\mathcal{C}(A_i)$. Beginning now with A_1 and A_2, the two lists $\mathcal{C}(A_1)$ and $\mathcal{C}(A_2)$ are compared, and the set of all pairs of coordinates (a_1, a_2) that satisfy the given distance constraints (l_1, u_1) at the appropriate level is constructed. (Note that a unique a_1 can satisfy the requirements with different a_2, and vice-versa). $\mathcal{C}(A_2)$ is now purged from the irrelevant coordinates, and the procedure is repeated with A_2 and A_3, resulting in the set $\{(a_1, a_2, a_3)\}$ of partial solutions of (1.1). Finally, when the last keyword A_m is processed in this way, we have the required set of solutions.

Note that it is not really necessary to always begin the processing with the first given keyword A_1 in (1.1), going all the way in a left-to-right mode. In some cases, it might be more efficient to begin it with a different keyword A_j, and to proceed with the other keywords in some specified order.

The main drawback of the inverted files approach is its huge overhead: the size of the concordance is comparable to that of the text itself and sometimes larger. For the intermediate range, a popular technique is based on assigning *signatures* to text fragments and to individual words. The signatures are then transformed into a set of bitmaps, on which Boolean operations, induced by the structure of the query, are performed. The idea is first to effectively reduce the size of the database by removing from consideration segments that cannot possibly satisfy the request, then to use pattern matching techniques to process the query, but only over the—hopefully small—remaining part of the database [4]. For systems supporting retrieval only at the document level, a different approach to query processing might be useful. The idea is to replace the concordance of a system with ℓ documents by a set of *bit-maps* of fixed length ℓ. Given some fixed ordering of the documents, a bit-map $B(W)$ is constructed for every distinct word W of the database, where the i-th bit of $B(W)$ is 1 if W occurs in the i-th document, and is 0 otherwise. Processing queries then reduces to performing logical OR/AND operations on binary sequences, which is easily done on most machines, instead of merge/collate operations on more general sequences. Davis & Lin [5] were apparently the first to propose the use of bit-maps for secondary key retrieval. It would be wasteful to store the bit-maps in their original form, since they are usually very sparse (the great majority of the words appear in very few documents), and we shall review various methods for the compression of such large sparse bit-vectors. However, the concordance can be dropped only if *all* the information we need is kept in the bit-maps. Hence, if we wish to extend this approach to systems supporting queries also at the paragraph, sentence or word-level, the length of each map must equal the number of paragraphs, sentences or words respectively, a clearly infeasible scheme for large systems. Moreover, the processing of distance constraints is hard to implement with such a data structure.

In [6], a method is presented in which, basically, the concordance and bit-map approaches are combined. At the cost of marginally expanding the inverted-files' structure, compressed bit-maps are *added* to the system; these maps give *partial* information on the location of the different words in the text and their distribution. This approach is described in more detail in Section 5.

Most of the techniques below were tested on two real-life full-text information retrieval systems, both using the inverted files approach. The one is the *Trésor de la Langue Française* (TLF) [7], a database of 680 MB of French language texts (112 million words) made up of a variety of complete documents including novels, short stories, poetry and essays, by many different authors. The bulk of the texts are from the 17th through 20th centuries, although smaller databases include texts from the 16th century and earlier. The other system is the *Responsa Retrieval Project* (RRP) [8], 350 MB of Hebrew and Aramaic texts (60 million words) written over the past ten centuries. For the sake of conciseness, detailed experimental results have been omitted throughout.

Table 1.1 shows roughly what one can expect from applying compression methods to the various files of a full-text retrieval system. The numbers correspond to TLF. Various smaller auxiliary files are not mentioned here, including grammatical files, thesauri, etc.

For the given example, the overall size of the system, which was close to two Gigabytes, could be reduced to fit onto a single CD-Rom.

The organization of this chapter is as follows. The subsequent sections consider, in turn, compression techniques for the file types mentioned above, namely, the text, dictionaries, concordances and bitmaps. For text compression, we first shortly review some background material. While concentrating on Huffman coding and related techniques, arithmetic coding

Table 1.1: *Files in a full-text system*

File	full size	compressed size	compression
Text	700 MB	245 MB	65%
Dictionary	30 MB	18 MB	40%
Concordance	400 MB	240 MB	40%
Bitmaps	800 MB	40 MB	95%
Total		543 MB	

and dictionary based text compression are also mentioned. For Huffman coding, we focus in particular on techniques allowing fast decoding, since decoding is more important than encoding in an Information Retrieval environment. For dictionary and concordance compression the prefix omission method and various variants are suggested. Finally, we describe the usefulness of bitmaps for the enhancement of IR systems and then show how these large structures may in fact be stored quite efficiently.

The choice of the methods to be described is not meant to be exhaustive. It is a blend of techniques which reflect the personal taste of the author rather than some well established core curriculum in Information Retrieval and Data Compression. The interested reader will find pointers to further details in the appended references.

2. TEXT COMPRESSION

We are primarily concerned with information retrieval, therefore this section will be devoted to text compression, as the text is still the heart of any large full-text IR system. We refer to text written in some natural language, using a fixed set of letters called an *alphabet*. It should however be noted that the methods below are not restricted to textual data alone, and are in fact applicable to any kind of file. For the ease of discourse, we shall still refer to texts and characters, but these terms should not be understood in their restrictive sense.

Whatever text of other file we wish to store, our computers insist on talking only binary, which forces us to transform the data using some binary *encoding*. The resulting set of elements, called *codewords*, each corresponding to one of the characters of the alphabet, is called a *code*. The most popular and easy to use codes are *fixed length* codes, for which all the codewords consist of the same number of bits. One of the best known fixed length codes is the American Standard Code for Information Interchange (ASCII), for which each codeword is one byte (eight bits) long, providing for the encoding of $2^8 = 256$ different elements.

A fixed length code has many advantages, most obviously, the encoding and decoding processes are straightforward. Encoding is performed by concatenating the codewords corresponding to the characters of the message, decoding is done by breaking the encoded string into blocks of the given size, and then using a decoding table to translate the codewords back into the characters they represent. For example, the ASCII representation of the word `Text` is

$$01010100011001010111100001110100,$$

which can be broken into

$$01010100 \,\big|\, 01100101 \,\big|\, 01111000 \,\big|\, 01110100.$$

From the compression point of view, such a code may be wasteful. A first attempt to reduce the space of an ASCII encoding is to note that if the actual character set used is of size n, only $\lceil \log_2 n \rceil$ bits are needed for each codeword. Therefore a text using only the 26 letters of the English alphabet (plus up to six special characters, such as space, period, comma, etc.) could be encoded using just five bits per codeword, saving already 37.5%. But even for larger alphabets an improvement is possible if the frequency of occurrence of the different characters is taken into account.

As is well-known, not all the letters appear with the same probability in natural language texts. For English, `E`, `T` and `A` are the most frequent, appearing about 12%, 10% and 8% respectively, while `J`, `Q` and `Z` occur each with probability less than 0.1%. Similar phenomena can be noted in other languages. The skewness of the frequency distributions can be exploited if one is ready to abandon the convenience of fixed length codes, and trade processing ease for better compression by allowing the codewords to have *variable length.* It is then easy to see that one may gain by assigning shorter codewords to the more frequent characters, even at the price of encoding the rare characters by longer strings, as long as the *average* codeword length is reduced. Encoding is just as simple as with fixed length codes and still consists in concatenating the codeword strings. There are however a few technical problems concerning the decoding that have to be dealt with.

A *code* has been defined above as a set of codewords, which are binary strings. But not every set of strings gives a useful code. Consider, for example, the four codewords in column (a) of Figure 2.1 If a string of 0's is given, it is easily recognized as a sequence of `A`'s. Similarly, the string 010101 can only be parsed as `BBB`. However, the string 010110 has two possible interpretations: $0 \,\big|\, 1011 \,\big|\, 0 =$ `ADA`, or $01 \,\big|\, 0 \,\big|\, 110 =$ `BAC`. This situation is intolerable, because it violates our basic premiss of reversibility of the encoding process. We shall thus restrict attention to codes for which *every* binary string obtained by concatenating codewords can be parsed only in one way, namely into the original sequence of codewords. Such codes are called *uniquely decipherable* (UD).

At first sight, it seems difficult to decide whether a code is UD or not, because infinitely many potential concatenations have to be checked. Nevertheless, efficient algorithms solving the problem do exist [9]. A necessary, albeit not sufficient, condition for a code to be UD is that its codewords should not be too short. A precise condition has been found by MacMillan [10]: any binary UD code with codewords lengths $\{\ell_1, \ldots, \ell_n\}$ satisfies

$$\sum_{i=1}^{n} 2^{-\ell_i} \le 1. \qquad (2.1)$$

`A`	0	`A`	11	`A`	11	`A`	1
`B`	01	`B`	110	`B`	011	`B`	00
`C`	110	`C`	1100	`C`	0011	`C`	010
`D`	1011	`D`	1101	`D`	1011	`D`	0110
		`E`	11000	`E`	00011	`E`	0111
Non-UD		UD non-prefix		prefix non-complete		complete	
(a)		(b)		(c)		(d)	

Figure 2.1: *Examples of codes*

For example, referring to the four codes of Figure 2.1, the sum is 0.9375, 0.53125, 0.53125 and 1 for codes (a) to (d) respectively. Case (a) is also an example showing that the condition is not sufficient.

But even if a code is UD, the decoding of certain strings may not be so easy. The code in column (b) of Figure 2.1 is UD, but consider the encoded string 11011111110: a first attempt to parse it as 110 | 11 | 11 | 11 | 10 = `BAAA`10 would fail, because the tail 10 is not a codeword; hence only when trying to decode the fifth codeword do we realize that the first one is not correct, and that the parsing should rather be 1101 | 11 | 11 | 110 = `DAAB`. In this case, a codeword is not immediately recognized as soon as all its bits are read, but only after a certain *delay*. There are codes for which this delay never exceeds a certain fixed number of bits, but the example above is easily extended to show that the delay for the given code is unbounded.

We would like to be able to recognize a codeword as soon as all its bits are processed, that is, with no delay at all; such codes are called *instantaneous*. A special class of instantaneous codes is known as the class of prefix codes: a code is said to have the *prefix property*, and is hence called a *prefix code*, if none of its codewords is a prefix of any other. It is unfortunate that this definition is misleading (shouldn't such a code be rather called a non-prefix code?), but it is widespread and therefore we shall keep it. For example, the code in Figure 2.1(a) is not prefix because the codeword for `A` (0) is a prefix of the codeword for `B` (01). Similarly, the code in (b) is not prefix, since all the codewords start with 11, which is the codeword for `A`. On the other hand, codes (c) and (d) are prefix.

It is easy to see that any prefix code is instantaneous and therefore UD. Suppose that while scanning the encoded string for decoding, a codeword x has been detected. In that case, there is no ambiguity as in the example above for code (b), because if there were another possible interpretation y which can be detected later, it would imply that x is a prefix of y, contradicting the prefix property.

In our search for good codes, we shall henceforth concentrate on prefix codes. In fact, we incur no loss by this restriction, even though the set of prefix codes is a proper subset of the UD codes: it can be shown that given any UD code whose codeword lengths are $\{\ell_1, \ldots, \ell_n\}$, one can construct a prefix code with the same set of codeword lengths [11]. As example, note that the prefix code (c) has the same codeword lengths as code (b). In this special case, (c)'s codewords are obtained from those of code (b) by reversing the strings; now every codeword terminates in 11, and the substring 11 occurs only as suffix of any codeword, thus no codeword can be the proper prefix of any other. Incidently, this also shows that code (b), which is not prefix, is nevertheless UD.

There is a natural one-to-one correspondence between binary prefix codes and binary trees. Let us assign labels to the edges and vertices of a binary tree in the following way:

- every edge pointing to a left child is assigned the label 0, and every edge pointing to a right child is assigned the label 1;
- the root of the tree is assigned the empty string Λ;
- every vertex v of the tree below the root is assigned a binary string which is obtained by concatenating the labels on the edges of the path leading from the root to vertex v.

It follows from the construction that the string associated with vertex v is a prefix of the string associated with vertex w if and only if v is a vertex on the path from the root to w. Thus the set of strings associated with the *leaves* of any binary tree satisfies the prefix property and may be considered as a prefix code. Conversely, given any prefix code, one can easily construct the corresponding binary tree. For example, the tree corresponding to the code $\{11, 101, 001, 000\}$ is depicted in Figure 2.2.

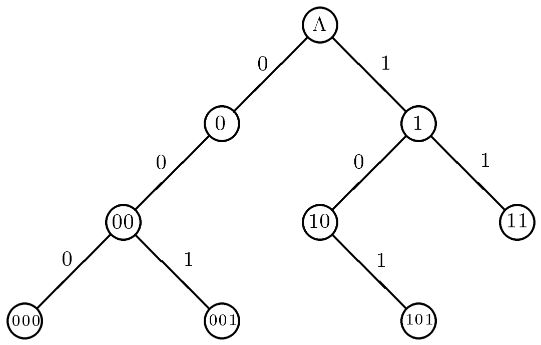


FIGURE 2.2: *Tree corresponding to code* $\{11, 101, 001, 000\}$

The tree corresponding to a code is a convenient tool for decompression. One starts with a pointer to the root and another one to the encoded string, which acts as a guide for the traversal of the tree. While scanning the encoded string from left to right, the tree-pointer is updated to point to the left, resp. right, child of the current node, if the next bit of the encoded string is a 0, resp. a 1. If a leaf of the tree is reached, a codeword has been detected, it is sent to the output and the tree-pointer is reset to point to the root.

Note that not all the vertices of the tree in Figure 2.2 have two children. From the compression point of view, this is a waste, because we could, in that case, replace certain codewords by shorter ones, without violating the prefix property, i.e., build another UD code with strictly smaller average codeword length. For example, the node labeled 10 has only a right child, so the codeword 101 could be replaced by 10; similarly, the vertex labeled 0 has only a left child, so the codewords 000 and 001 could be replaced by 00 and 01, respectively. A tree for which all internal vertices have two children is called a *complete* tree, and accordingly, the corresponding code is called a complete code. A code is complete if and only if the lengths $\{\ell_i\}$ of its codewords satisfy equation (2.1) with equality, i.e., $\sum_{i=1}^{n} 2^{-\ell_i} = 1$.

2.1 Huffman Coding

To summarize what we have seen so far, we have restricted the class of codes under consideration in several steps. Starting from general UD codes, passing to instantaneous and prefix codes and finally to complete prefix codes, since we are interested in good compression performance. The general problem can thus be stated as follows: we are given a set of n non-negative weights $\{w_1, \ldots, w_n\}$, which are the frequencies of occurrence of the letters of some alphabet. The problem is to generate a complete binary variable-length prefix code, consisting of codewords with lengths ℓ_i bits, $1 \le i \le n$, with optimal compression capabilities, i.e., such that the *total length* of the encoded text

$$\sum_{i=1}^{n} w_i \ell_i \tag{2.2}$$

is minimized. It is sometimes convenient to redefine the problem in terms of relative frequencies. Let $W = \sum_{i=1}^{n} w_i$ be the total number of characters in the text, one can then define $p_i = w_i/W$ as the *probability of occurrence* of the i-th letter. The problem is then equivalent to minimizing the *average codeword length* $\sum_{i=1}^{n} p_i \ell_i$.

Let us for a moment forget about the interpretation of the ℓ_i as codeword lengths, and try to solve the minimization problem analytically without restricting the ℓ_i to be integers, but still keeping the constraint that they must satisfy the McMillan equality $\sum_{i=1}^{n} 2^{-\ell_i} = 1$. To find the set of ℓ_i's minimizing (2.2), one can use Langrange multipliers. Define a function $L(\ell_1, \ldots, \ell_n)$ of n variables and with a parameter λ by

$$L(\ell_1, \ldots, \ell_n) = \sum_{i=1}^{n} w_i \ell_i - \lambda \left(\sum_{i=1}^{n} 2^{-\ell_i} - 1 \right),$$

and look for local extrema by setting the partial derivatives to zero:

$$\frac{\partial L}{\partial \ell_i} = w_i + \lambda\, 2^{-\ell_i} \ln 2 = 0,$$

which yields

$$\ell_i = -\log_2 \left(\frac{w_i}{-\lambda \ln 2} \right). \tag{2.3}$$

To find the constant λ, substitute the values for ℓ_i derived in (2.3) in the McMillan equality:

$$1 = \sum_{i=1}^{n} 2^{-\ell_i} = \frac{1}{-\lambda \ln 2} \sum_{i=1}^{n} w_i = \frac{W}{-\lambda \ln 2},$$

from which one can derive $\lambda = -W/\ln 2$. Plugging this value back into (2.3), one finally gets

$$\ell_i = -\log_2 \left(\frac{w_i}{W} \right) = -\log_2 p_i.$$

This quantity is known as *the information content* of a symbol with probability p_i, and it represents the minimal number of bits in which the symbol could be coded. Note that this number is not necessarily an integer. Returning to the sum in (2.2), we may therefore conclude that the lower limit of the total size of the encoded file is given by

$$-\sum_{i=1}^{n} w_i \log_2 p_i = W\left(-\sum_{i=1}^{n} p_i \log_2 p_i\right). \tag{2.4}$$

The quantity $H = -\sum_{i=1}^{n} p_i \log_2 p_i$ has been defined by Shannon [12] as the *entropy* of the probability distribution $\{p_1, \ldots, p_n\}$, and it gives a lower bound to the weighted average codeword length.

In 1952, Huffman [13] proposed the following algorithm which solves the problem.

1. If $n = 1$, the codeword corresponding to the only weight is the null-string;   return.
2. Let w_1 and w_2, without loss of generality, be the two smallest weights.
3. Solve the problem recursively for the $n-1$ weights $w_1 + w_2, w_3, \ldots, w_n$;
   let α be the codeword assigned to the weight $w_1 + w_2$.
4. The code for the n weights is obtained from the code for $n-1$ weights generated in point 3 by replacing α by the two codewords $\alpha 0$ and $\alpha 1$;   return.

In the straightforward implementation, the weights are first sorted and then every weight obtained by combining the two which are currently the smallest, is inserted in its proper place in the sequence so as to maintain order. This yields an $O(n^2)$ time complexity. One can reduce the time complexity to $O(n \log n)$ by using two queues, the one, Q_1, containing the original elements, the other, Q_2, the newly created combined elements. At each step, the two smallest elements in $Q_1 \cup Q_2$ are combined and the resulting new element is inserted at the end of Q_2, which remains in order [14].

Theorem. *Huffman's algorithm yields an optimal code.*

Proof: By induction on the number of elements n. For $n = 2$, there is only one complete binary prefix code, which therefore is optimal, namely $\{0, 1\}$; this is also a Huffman code, regardless of the weights w_1 and w_2.

Assume the truth of the Theorem for $n-1$. Let T_1 be an optimal tree for $\{w_1, \ldots, w_n\}$, with ACL $M_1 = \sum_{i=1}^{n} w_i l_i$.

Claim 1: There are at least two elements on the lowest level of T_1.

Proof: Suppose there is only one such element and let $\gamma = a_1 \cdots a_m$ be the corresponding binary codeword. Then by replacing γ by $a_1 \cdots a_{m-1}$ (i.e., dropping the last bit) the resulting code would still be prefix, and the ACL would be smaller, in contradiction with T_1's optimality.

Claim 2: The codewords c_1 and c_2 corresponding to the smallest weights w_1 and w_2 have maximal length (the nodes are on the lowest level in T_1).

Proof: Suppose the element with weight w_2 is on level m, which is not the lowest level ℓ. Then there is an element with weight $w_x > w_2$ at level ℓ. Thus the tree obtained by switching w_x

with w_2 has an ACL of

$$M_1 - w_x\ell - w_2m + w_xm + w_2\ell < M_1,$$

which is impossible since T_1 is optimal.

Claim 3: Without loss of generality one can assume that the smallest weights w_1 and w_2 correspond to sibling nodes in T_1.

Proof: Otherwise one could switch elements without changing the ACL.

Consider the tree T_2 obtained from T_1 by replacing the sibling nodes corresponding to w_1 and w_2 by their common parent node α, to which the weight $w_1 + w_2$ is assigned. Thus the ACL for T_2 is $M_2 = M_1 - (w_1 + w_2)$.

Claim 4: T_2 is optimal for the weights $(w_1 + w_2), w_3, \ldots, w_n$.

Proof: If not, let T_3 be a better tree with $M_3 < M_2$. Let β be the node in T_3 corresponding to $(w_1 + w_2)$. Consider the tree T_4 obtained from T_3 by splitting β and assigning the weight w_1 to β's left child and w_2 to its right child. Then the ACL for T_4 is

$$M_4 = M_3 + (w_1 + w_2) < M_2 + (w_1 + w_2) = M_1,$$

which is impossible, since T_4 is a tree for n elements with weights $w_1, \ldots, w_n$ and T_1 is optimal among all those trees.

Using the inductive assumption, T_2, which is an optimal tree for $n - 1$ elements, has the same ACL as the Huffman tree for these weights. However, the Huffman tree for $w_1, \ldots, w_n$ is obtained from the Huffman tree for $(w_1 + w_2), w_3, \ldots, w_n$ in the same way as T_1 is obtained from T_2. Thus the Huffman tree for the n elements has the same ACL as T_1, hence it is optimal. ∎

2.2 Huffman Coding without Bit-Manipulations

In many applications, compression is by far not as frequent as decompression. In particular, in the context of static IR systems, compression is done only once (when building the database), whereas decompression directly affects the response time for on-line queries. We are thus more concerned with a good *decoding* procedure. In spite of their optimality, Huffman codes are not always popular with programmers as they require bit-manipulations and are thus not suitable for smooth programming and efficient implementation in most high-level languages.

This section presents decoding routines that directly process only bit-blocks of fixed and convenient size (typically, but not necessarily, integral bytes), making it therefore faster and better adapted to high-level languages programming, while still being efficient in terms of space requirements. In principle, byte-decoding can be achieved either by using specially built tables to isolate each bit of the input into a corresponding byte or by extracting the required bits while simulating shift operations.

2.2.1 Eliminating the reference to bits

We are given an alphabet Σ, the elements of which are called *letters*, and a message ($\equiv$ sequence of elements of Σ) to be compressed, using variable-length codes. Let L denote the set of N items to be encoded. Often $L = \Sigma$, but we do not restrict the codewords necessarily to represent single letters of Σ. Indeed, the elements of L can be pairs, triplets or any n-grams of letters, they can represent words of a natural language, and they can finally form a set of items of completely different nature, provided that there is an unambiguous way to decompose a given file into these items (see for example [15]). We call L an *alphabet* and its elements *characters*, where these terms should be understood in a broad sense. We thus include also in our discussion applications where N, the size of the "alphabet", can be fairly large.

We begin by compressing L using the variable-length Huffman codewords of its different characters, as computed by the conventional Huffman algorithm. We now partition the resulting bit-string into k-bit blocks, where k is chosen so as to make the processing of k-bit blocks, with the particular machine and high-level language at hand, easy and natural. Clearly, the boundaries of these blocks do not necessarily coincide with those of the codewords: a k-bit block may contain several codewords, and an codeword may be split into two (or more) adjacent k-bit blocks. As an example, let $L = \{\texttt{A},\texttt{B},\texttt{C},\texttt{D}\}$, with codewords $\{0, 11, 100, 101\}$ respectively, and choose $k = 3$. Consider the following input string, its coding and the coding's partition into 3-bit blocks:

$$\overbrace{0}^{\texttt{A}}\quad\overbrace{0}^{\texttt{A}}\quad\overbrace{1\quad 1}^{\texttt{B}}\quad\overbrace{1\quad 0\quad 1}^{\texttt{D}}\quad\overbrace{1\quad 1}^{\texttt{B}}$$

$$\underbrace{0\quad 0\quad 1}_{1}\quad\underbrace{1\quad 1\quad 0}_{6}\quad\underbrace{1\quad 1\quad 1}_{7}$$

The last line gives the integer value $0 \leq i < 2^3$ of the block.

The basic idea for all the methods is to use these k-bit blocks, which can be regarded as the binary representation of integers, as *indices* to some tables which are prepared in advance in the preprocessing stage.

In this section we first describe two straightforward — albeit not very efficient — methods for implementing this idea.

For the first method, we use a table $\mathcal{B}$ of 2^k rows and k columns. In fact, $\mathcal{B}$ will contain only zeros and ones, but as we want to avoid bit-manipulations, we shall use one byte for each of the $k2^k$ elements of this matrix. Let $i = I_1 \cdots I_k$ be the binary representation of length k (with leading zeros) of i, for $0 \leq i < 2^k$, then $\mathcal{B}(i, j) = I_j$, for $1 \leq j \leq k$; in other words, the i-th line of B contains the binary representation of i, one bit per byte. The matrix $\mathcal{B}$ will be used to decompose the input string into individual bits, without any bit-manipulation. Figure 2.3(a) depicts the matrix $\mathcal{B}$ for $k = 3$.

The values 0 or 1 extracted from $\mathcal{B}$ are used to decode the input, using the Huffman tree of the given alphabet. The Huffman tree of the alphabet L of our small example is in Figure 2.4(a).

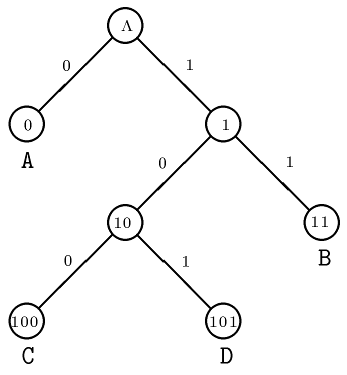
$\mathcal{B}$	1	2	3
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

(a)

$\mathcal{S}$	1	2
0	0	0
1	2	0
2	4	0
3	6	0
4	0	1
5	2	1
6	4	1
7	6	1

(b)

FIGURE 2.3: *Tables for Huffman decoding*



(a) Tree form

$\mathcal{H}$	0	1
0	-A	1
1	2	-B
2	-C	-D

(b) Table form

FIGURE 2.4: *Example of Huffman code*

A Huffman tree with N leaves (and $N-1$ internal nodes) can be kept as a table $\mathcal{H}$ with $N-1$ rows (one for each internal node) and two columns. The internal nodes are numbered from 0 to $N-2$ in arbitrary order, but for convenience the root will always be numbered zero. For example in Figure 2.4(a), the indices of the internal nodes containing Λ, 1 and 10 will be 0, 1 and 2 respectively. The two elements stored in the i-th row of table $\mathcal{H}$ are the left and right children of the internal node indexed i. Each child can be either another internal node, in which case its index is stored, or a leaf, corresponding to one of the characters of the alphabet, in which case this character is stored. We thus need an additional bit per element, indicating whether it is an internal node or a leaf, but generally, one can use the sign-bit for that purpose: if the element is positive, it represents the index of an internal node; if it is negative, its absolute value is the representation of a character. Figure 2.4(b) shows the table $\mathcal{H}$ corresponding to the Huffman tree of Figure 2.4(a). The Huffman decoding routine can then be formulated as follows:

Byte Decoding algorithm

```
$ind \leftarrow 0$        [ pointer to table $H$]
repeat
        $n \leftarrow$ integer value of next input block
        for  $j = 1$  to  $k$
                $newind \leftarrow H\,(ind, B(n, j))$        [ left or right child of current node]
                if  $newind > 0$  then  $ind \leftarrow newind$
                else
                        output($-newind$)
                        $ind \leftarrow 0$
        end
until  input is exhausted
```

Another possibility is to replace table $\mathcal{B}$ by the following table $\mathcal{S}$, again with 2^k rows, but only two columns. For $0 \leq i < 2^k$, $\mathcal{S}(i, 1)$ will contain $2i \bmod 2^k$, and $\mathcal{S}(i, 2)$ will contain the leftmost bit of the k-bit binary representation of i. In the algorithm, the assignment to $newind$ has to be replaced by

```
$newind \leftarrow H\,(ind, S(n, 2))$
$n \leftarrow \mathcal{S}(n, 1)$
```

The first statement extracts the leftmost bit and the second statement shifts the k-bit block by one bit to the left. Figure 2.3(b) shows table $\mathcal{S}$ for $k = 3$. Hence we have reduced the space needed for the tables from $k2^k + 2(N-1)$ to $2^{k+1} + 2(N-1)$, but now there are three table accesses for every bit of the input, instead of only two accesses for the first method.

Although there is no reference to bits in these algorithms and their programming is straightforward, the number of table accesses makes their efficiency rather doubtful; their only advantage is that their space requirements are linear in N (k is a constant), while for all other time-efficient variants to be presented below, space is at least $\Omega(N \log N)$. However, for these first two methods, the term 2^k of the space complexity is dominant for small N, so that they can be justified — if at all — only for rather large N.

2.2.2 Partial-decoding tables

Recall that our goal is to permit a block-per-block processing of the input string for some fixed block-size k. Efficient decoding under these conditions is made possible by using a set of m auxiliary tables, which are prepared in advance for every given Huffman code, whereas tables $\mathcal{B}$ and $\mathcal{S}$ above were independent of the character distribution.

The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (W, j), where W is a sequence of characters and j $(0 \leq j < m)$ is the index of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence W of characters from the k-bit block representing the integer i (W may be empty when there are codewords of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P of more than one codeword; j will then be the index of the table corresponding to that prefix (if $P = \Lambda$, then $j = 0$). Table number j is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P to the binary representation of i. We thus need a table for every possible proper prefix of the given codewords; the number of these prefixes is obviously equal to the number of internal nodes of the appropriate Huffman-tree (the root corresponding to the empty string and the leaves corresponding to the codewords), so that $m = N - 1$.

More formally, let P_j, $0 \leq j < N - 1$, be an enumeration of all the proper prefixes of the codewords (no special relationship needs to exist between j and P_j, except for the fact that $P_0 = \Lambda$). In table j corresponding to P_j, the i-th entry, $T(j, i)$, is defined as follows: let B be the bit-string composed of the juxtaposition of P_j to the left of the k-bit binary representation of i. Let W be the (possibly empty) longest sequence of characters that can be decoded from B, and P_ℓ the remaining undecipherable bits of B; then $T(j, i) = (W, \ell)$.

Entry	Pattern	Table 0		Table 1		Table 2	
	for Table 0	W	ℓ	W	ℓ	W	ℓ
0	000	AAA	0	CA	0	CAA	0
1	001	AA	1	C	1	CA	1
2	010	A	2	DA	0	C	2
3	011	AB	0	D	1	CB	0
4	100	C	0	BAA	0	DAA	0
5	101	D	0	BA	1	DA	1
6	110	BA	0	B	2	D	2
7	111	B	1	BB	0	DB	0

FIGURE 2.5: *Partial decoding tables*

Referring again to the simple example given above, there are 3 possible proper prefixes: Λ,1,10, hence 3 corresponding tables indexed 0,1,2 respectively, and these are given in Figure 2.5. The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1 and 2 are obtained by prefixing '1', respectively '10', to the strings in 'Pattern'.

For the input example given above, we first access Table 0 at entry 1, which yields the output string AA; Table 1 is then used with entry 6, giving the output B; finally Table 2 at entry 7 gives output DB.

The utterly simple decoding subroutine (for the general case) is as follows ($M(i)$ denotes the i-th block of the input stream, j is the index of the currently used table and $T(j, \ell)$ is the ℓ-th entry of table j):

Basic Decoding Algorithm

```
$j \leftarrow 0$
for   $i \leftarrow 1$   to   length of input   do
        (output, $j$) $\leftarrow T(j, M(i))$
end
```

As mentioned before, the choice of k is largely governed by the machine-word structure and the high-level language architecture. A natural choice in most cases would be $k = 8$, corresponding to a byte context, but $k = 4$ (half-byte) or $k = 16$ (half-word) are also conceivable. The larger is k, the greater is the number of characters that can be decoded in a single iteration, thus transferring a substantial part of the decoding time to the preprocessing stage. The size of the tables however grows exponentially with k, and with every entry occupying (for $N \leq 256$ and $k = 8$) 1 to 8 bytes, each table may require between $1K$ and $2K$ bytes of internal memory. For $N > 256$, we need more than one byte for the representation of a character, so that the size of a table will be even larger, and for larger alphabets these storage requirements may become prohibitive. We now develop an approach that can help reduce the number of required tables and their size.

2.2.3 Reducing the number of tables: binary forests

The storage space needed by the partial decoding tables can be reduced by relaxing somewhat the approach of the previous section, and using the conventional Huffman decoding algorithm no more than once for every block, while still processing only k-bit blocks. This is done by redefining the tables and adding some new data-structures.

Let us suppose, just for a moment, that after deciphering a given block B of the input that contains a "remainder" P (which is a prefix of a certain codeword), we are somehow able to determine the correct complement of P and its length ℓ, and accordingly its corresponding encoded character. More precisely, since an codeword can extend into more than two blocks, ℓ will be the length of the complement of P in the next k-bit block which contains also other codewords, hence $0 \leq \ell < k$. In the next iteration (decoding of the next k-bit block which was not yet entirely deciphered), table number ℓ will be used, which is similar to table 0, but *ignores* the first ℓ bits of the corresponding entry, instead of *prefixing* P to this entry as in the previou section.

Therefore the number of tables reduces from $N - 1$ (about 30 in a typical single-letter natural-language case, or 700–900 if we use letter pairs) to only k (8 or 16 in a typical byte or half-word context), where entry i in table ℓ, $0 \leq \ell < k$, contains the decoding of the $k - \ell$ rightmost bits of the binary representation of i. It is clear, however, that Table 1 contains two exactly equal halves, and in general table ℓ $(0 \leq \ell < k)$ consists of 2^ℓ identical parts. Retaining then in each table only the first $2^{k-\ell}$ entries, we are able to compress the needed k tables into the size of only two tables. The entries of the tables are again of the form (W, j); note however that j is not an index to the next table, but an identifier of the remainder P;

it is only after finding the correct complement of P and its length ℓ that we can access the right Table ℓ.

For the same example as before one obtains the tables of Figure 2.6, where table t decodes the bit-strings given in 'Pattern', but ignoring the t leftmost bits, $t = 0, 1, 2$, and $l = 0, 1, 2$ corresponds respectively to the proper prefixes $\Lambda, 1, 10$.

Entry	Pattern	Table 0		Table 1		Table 2	
	for Table 0	W	ℓ	W	ℓ	W	ℓ
0	000	AAA	0	AA	0	A	0
1	001	AA	1	A	1	–	1
2	010	A	2	–	2		
3	011	AB	0	B	0		
4	100	C	0				
5	101	D	0				
6	110	BA	0				
7	111	B	1				

FIGURE 2.6: *Sub-string translate tables*

The algorithm will be completed if we can find a method to identify the codeword corresponding to the remainder of a given input block, using of course the following input block(s). We introduce the method through an example.

Figure 2.7 shows a typical Huffman tree H for an alphabet L of $N = 7$ characters. Assume now $k = 8$ and consider the following adjacent blocks of input: 00101101 00101101. The first block is decoded into the string BE and the remainder $P = 01$. Starting at the internal node containing 01 and following the first bits of the following block, we get the codeword C, and length $l = 2$ for the complement of P, so that Table 2 will be used when decoding the next block; ignoring the first 2 bits, this table translates the binary string 101101.

For the general case, let us for simplicity first assume that the depth of H, which is the length of the longest codeword, is bounded by k. Given the non-empty remainder P of the current input block, we must access the internal node corresponding to P, proceed downwards turning left (0) or right (1) as indicated by the first few bits of the next k-bit block, until we reach a leaf. This leaf contains the next character of the output. The number of edges traversed is the index of the table to be used in the next iteration.

Our goal is to simulate this procedure without having to follow a "bit-traversal" of the tree. The algorithm below uses a binary forest instead of the original Huffman tree H. For the sake of clarity, the construction of the forest is described in two steps.

First, replace H by $N - 2$ smaller trees H_i, which are induced by the proper sub-trees rooted at the internal nodes of H, corresponding to all non-empty proper prefixes of the codewords. The nodes of the trees of the forest contain binary strings: Λ for the roots, and for each other node v, a string obtained by concatenating the labels of the edges on the path

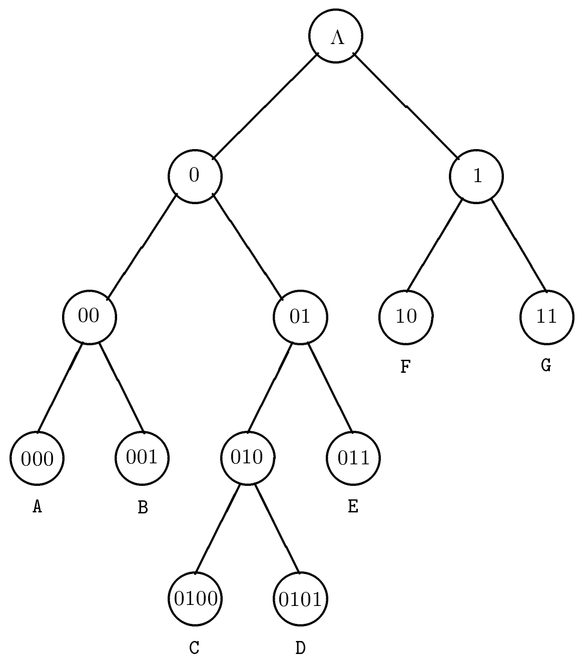


FIGURE 2.7: *The Huffman Tree H*

from the root to v, as in the Huffman tree, but padded at the right by zeroes so as to fill a k-bit block. In addition, each leaf contains also the corresponding decoded character. The string in node v is denoted by VAL(v). Figure 2.8 depicts the forest obtained from the tree of our example, where the pointer to each tree is symbolized by the corresponding proper prefix.

The idea is that the identifier of the remainder in an entry of the tables described above is in fact a pointer to the corresponding tree. The traversal of this tree is guided by the bits of the next k-bit block of the input, which can directly be compared with the contents of the nodes of the tree, as will be described below.

Consider now also the possibility of long codewords, which extend over several blocks. They correspond to long paths so that the depth of some trees in the forest may exceed k. During the traversal of a tree, passing from one level to the next lowest one is equivalent to advancing one bit in the input string. Hence when the depth exceeds k, all the bits of the current k-bit block were used, and we pass to the next block. Therefore the above definition of VAL(v) applies only to nodes on levels up to k; this definition is generalized to any node by: VAL(v) for a node v on level j, with $ik < j \leq (i+1)k$, $i \geq 0$, is the concatenation of the labels on the edges on the path from level ik to v.

In the second step, we compress the forest as could have been done with any Huffman tree. In such trees, every node has degree 0 or 2, i.e. they appear in pairs of siblings (except the root). For a pair of sibling-nodes (a, b), VAL(a) and VAL(b) differ only in the j-th bit, where j is the level of the pair (here and in what follows, the level of the root of a tree is 0), or more precisely, $j = (\text{level} - 1) \bmod k + 1$. In the compressed tree, every pair is represented by an unique node containing the VAL of the right node of the pair, the new root is the node obtained from the only pair in level 1, and the tree structure is induced by the non-compressed

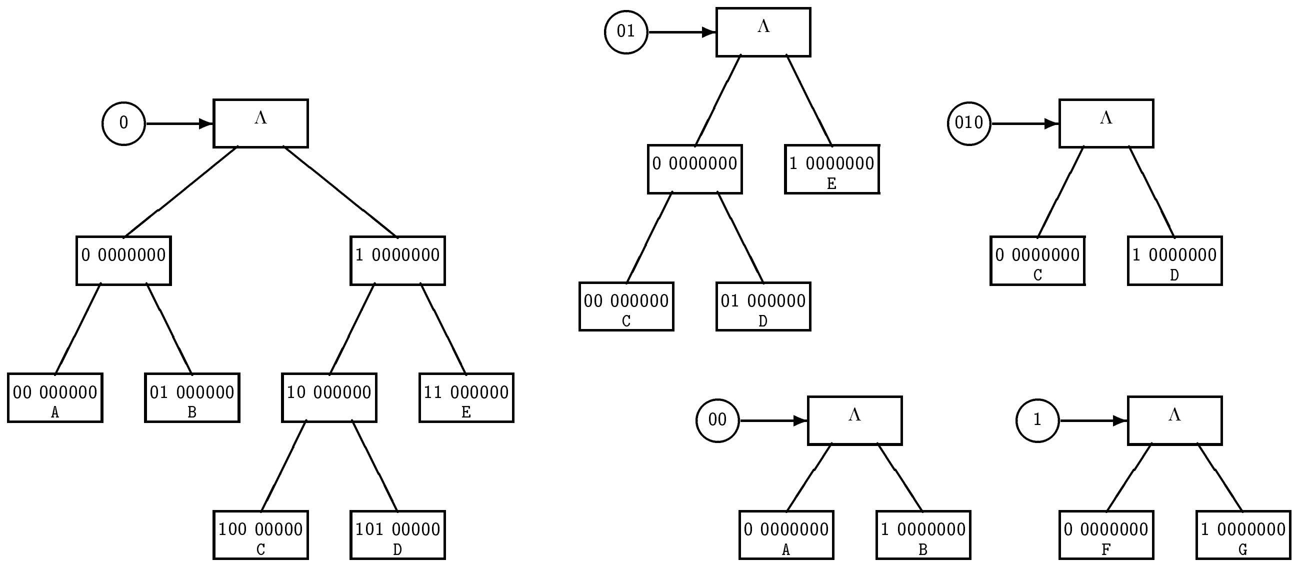


FIGURE 2.8: *Forest of proper prefixes*

tree. Thus a tree of ℓ nodes shrinks now to $(\ell - 1)/2$ nodes. Another way to look at this "compression" method is to take the tree of internal nodes, and store it in form of a table as was described in the previous section. We use here a tree-oriented vocabulary, but each tree can equivalently be implemented as a table. Figure 2.9 is the compressed form of the forest of Figure 2.8.

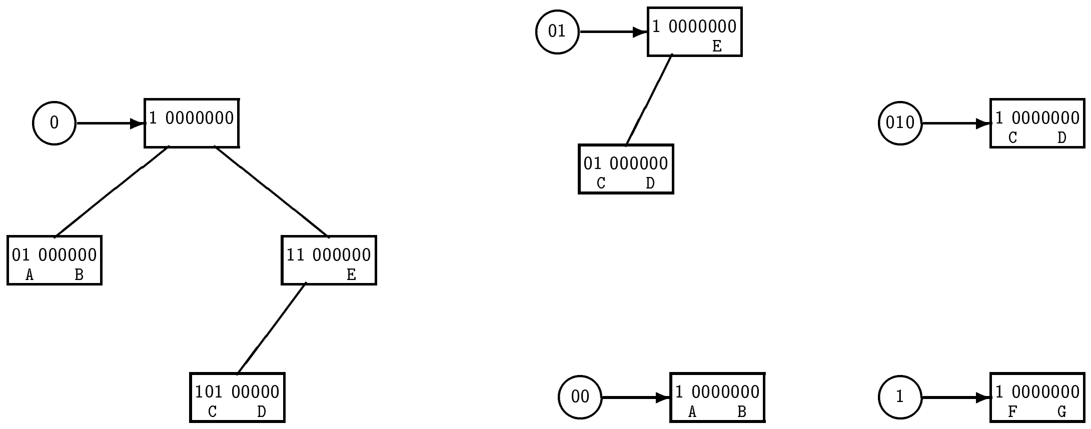


FIGURE 2.9: *Compressed forest*

We can now compare directly the values VAL stored in the nodes of the trees with the k-bit blocks of the Huffman encoded string. The VAL values have the following property: let v be a node on level j of one of the trees in the compressed forest, with $ik < j \le (i+1)k$, $i \ge 0$ as above, and let $I(B)$ be the integer value of the next k-bit block B. Then

$$I(B) < \text{VAL}(v) \qquad \text{if and only if} \qquad \text{bit } (1 + j \bmod k) \text{ of } B \text{ is } 0.$$

Thus after accessing one of the trees, the VAL of its root is compared with the next k-bit block B. If B, interpreted as a binary integer, is smaller, it must start with 0 and we turn left; if B is greater or equal, it must start with 1 and we turn right. These comparisons are repeated at the next levels, simulating the search for an element in a binary search tree [16, Section 6.2.2]. This leads to the modified algorithm below. Notations are like before, ROOT(t) points to the t-th tree of the forest, every node has three fields: VAL, a k-bit value, LEFT and RIGHT each of which is either a pointer to the next level or contains a character of the alphabet. When accessing table j, the index is taken modulo the size of the table, which is 2^{k-j}.

```
Revised Decoding Algorithm
$i \leftarrow 1$
$j \leftarrow 0$
repeat
    (output, tree-nbr) $\leftarrow T(j, S(i)$ mod $2^{k-j})$
    $i \leftarrow i + 1$
    j $\leftarrow 0$
    if  tree-nbr $\neq 0$  then   TRAVERSE ( ROOT(tree-nbr) )
until  input is exhausted
```

where the procedure TRAVERSE is defined by

```
TRAVERSE ( node )
    repeat
        if  $S(i) <$ VAL(node)  then
                    node $\leftarrow$ LEFT(node)
        else        node $\leftarrow$ RIGHT(node)
        if  node is a character $C$   then  output $C$
        $j \leftarrow j + 1$      [$j$ is the number of bits in $S(i)$ which are 'used up']
        if  $j = k$  then
            $j \leftarrow 0$
            $i \leftarrow i + 1$      [advance to next $k$-bit block]
    until  a character was output
end
```

Any node v of the original (compressed) Huffman tree H' generates several nodes in the forest, the number of which is equal to the level of v in H'. Hence the total number of nodes in the forest is exactly the *internal path length* of the original (uncompressed) Huffman tree H, as defined by Knuth [17]. This quantity is between $O(N \log N)$ (for a full binary tree) and $O(N^2)$ (for a degenerate tree), and at the average, with all possible shapes of Huffman trees equally likely, proportional to $N\sqrt{N}$.

Therefore even in the worst case, the space requirements are reasonable in most practical applications with small N. If, for large N and certain probability distributions, $O(N^2)$ is prohibitive, it is possible to keep the space of the forest bounded by $O(N \log N)$, if one agrees to abandon the optimality of the Huffman tree. This can be done by imposing a maximal length of $K = O(\log N)$ to the codewords. If K does not exceed the block-size k, the decoding algorithm can even be slightly simplified, since in the procedure TRAVERSE there is no need to check if the end of the block was reached. An other advantage of bounding the depth of the Huffman tree is that this tends to lengthen the shortest codeword. Since the number of characters stored at each entry in the partial-decoding tables is up to $1 + \lceil (k-1)/s \rceil$, where s is the length of the shortest codeword, this can reduce the space required to store each table. An algorithm for the construction of an optimal tree with bounded depth in time and space $O(KN)$ can be found in [18]. Nevertheless, it might often not seem worthwhile to spend so much efforts to obtain an *optimal* code of bounded length. As alternative one can use a procedure proposed in [19], which gives sub-optimal average codeword length, but uses less space and is much faster. Moreover, the codes constructed by this method are often very near to optimal.

2.2.4 Huffman codes with radix $r > 2$

The number of tables can also be reduced by the following simple variants which, similar to the variants with bounded codeword length, yield slightly lower compression factors than the methods described above. Let us apply the Huffman algorithm with radix r, $r > 2$, the details of which can be found in Huffman's original paper [13]. In such a variant, one combines at each step, except perhaps the first, the r smallest weights (rather than only the smallest two in the binary algorithm) and replaces them by their sum. The number of weights combined in the first step is chosen so that the number h of weights remaining after this step verifies $h \equiv 1 \pmod{r-1}$. In the corresponding r-ary tree, every internal node has r sons, except perhaps one on the next-to-lowest level of the tree which has between 2 and r sons. If we choose $r = 2^\ell$, we can encode the alphabet in a first stage using r different symbols; then every symbol is replaced by a binary code of ℓ bits. If in addition ℓ divides k, the "borders" of the k-bit blocks never split any ℓ-bit code. Hence in the partial-decoding tables, the possible remainders are sequences of one or more r-ary symbols. There is therefore again a correspondence between the possible remainders and the internal nodes of the r-ary Huffman tree, only that their number now decreased to $\lceil (n-1)/(r-1) \rceil$. Moreover, there may be some savings in the space needed for a specific table. As we saw before, the space for each table depends on the length s of the shortest codeword, so this can be k with the binary algorithm when $s = 1$, but at most $\lceil k/2 \rceil$ in the 4-ary case.

Due to the restrictions on the choice of r, there are only few possible values. For example, for $k = 8$, one could use a quaternary code ($r = 2^2$), where every code-word has an even number of bits and the number of tables is reduced by a factor of 3, or a hexadecimal code ($r = 2^4$), where the code-word length is a multiple of 4 and the number of tables is divided by 15. Note that for alphabets with $N \leq 31$, the hexadecimal code can be viewed as the classical method using "restricted variability" (see for example [20]: assign 4-bit encodings to the 15 most frequent characters and use the last 4-bit pattern as "escape character" to indicate that the actual character is encoded in the next 4 bits. Thus up to 16 least frequent characters have 8-bit encodings, all of which have their first 4 bits equal to the escape character.

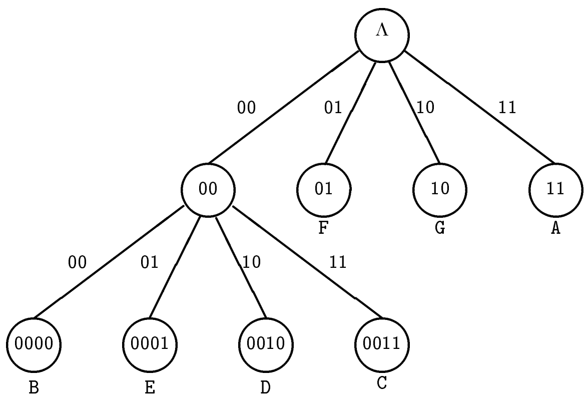


FIGURE 2.10: *Quaternary Huffman tree*

Referring to the Huffman tree given in Figure 2.7, suppose that a character corresponding to a leaf on level ℓ appears with probability $2^{-\ell}$, then the corresponding 2^2-ary tree is given in Figure 2.10. Note that the only proper prefixes of even length are Λ and 00, so that the number of tables dropped from 6 to 2.

However, with increasing r, compression will get worse, so that the right trade-off must be chosen according to the desired application.

2.3 Space Efficient Decoding of Huffman Codes

The data structures needed for the decoding of a Huffman encoded file (a Huffman tree or lookup table) are generally considered negligible overhead relative to large texts. However, not all texts are large, and if Huffman coding is applied in connection with a Markov model [21], the required Huffman forest may become itself a storage problem. Moreover, the "alphabet" to be encoded is not necessarily small, and may, e.g., consist of all the different words in the text, so that Huffman trees with thousands and even millions of nodes are not uncommon [22]. We try here to reduce the necessary internal memory space by devising efficient ways to encode these trees. In addition, the suggested data structure also allows a speed-up of the decompression process, by reducing the number of necessary bit comparisons.

2.3.1 Canonical Huffman codes

0	000
1	0010
2	0011
3	0100
4	01010
5	01011
6	01100
7	01101
8	011100
9	011101
10	011110
11	011111
12	100000
13	100001
14	100010
15	100011
16	1001000
17	1001001
18	1001010
19	1001011
...	...
29	1010101
30	1010110
31	10101110
32	10101111
33	10110000
...	...
61	11001100
62	11001101
63	110011100
64	110011101
...	...
124	111011001
125	111011010
126	1110110110
127	1110110111
...	...
198	1111111110
199	1111111111

FIGURE 2.11: *Canonical Huffman code for Zipf-200*

For a given probability distribution, there might be quite a large number of different Huffman trees, since interchanging the left and right subtrees of any internal node will result in a different tree whenever the two subtrees are different in structure, but the weighted average path length is not affected by such an interchange. There are often also other optimal trees, which cannot be obtained via Huffman's algorithm. One may thus choose one of the trees that has some additional properties. The preferred choice for many applications is the *canonical* tree, defined by Schwartz and Kallick [23], and recommended by many others (see, e.g., [24, 25]).

Denote by $(p_1, \ldots, p_n)$ the given probability distribution, where we assume that $p_1 \geq p_2 \geq \cdots \geq p_n$, and let ℓ_i be the length in bits of the codeword assigned by Huffman's procedure to the element with probability p_i, i.e., ℓ_i is the depth of the leaf corresponding to p_i in the Huffman tree. A tree is called canonical if, when scanning its leaves from left to right, they appear in non-decreasing order of their depth (or equivalently, in non-increasing order, as in [26]). The idea is that Huffman's algorithm is only used to generate the lengths $\{\ell_i\}$ of the codewords, rather than the codewords themselves; the latter are easily obtained as follows: the i-th codeword consists of the first ℓ_i bits immediately to the right of the "binary point" in the infinite binary expansion of $\sum_{j=1}^{i-1} 2^{-\ell_j}$, for $i = 1, \ldots, n$ [27]. Many properties of canonical codes are mentioned in [24, 28].

The following will be used as a running example in this section. Consider the probability distribution implied by Zipf's law, defined by the weights $p_i = 1/(i\,H_n)$, for $1 \leq i \leq n$, where $H_n = \sum_{j=1}^{n}(1/j)$ is the n-th harmonic number. This law is believed to govern the distribution of the most common words in a large natural language text [29]. A canonical code can be represented by the string $\langle n_1, n_2, \ldots, n_k \rangle$, called a *source*, where k denotes, here and below, the length of the longest codeword (the depth of the tree), and n_i is the number of codewords of length i, $i = 1, \ldots, k$. The source corresponding to Zipf's distribution for $n = 200$ is $\langle 0, 0, 1, 3, 4, 8, 15, 32, 63, 74 \rangle$. The code is depicted in Figure 2.11.

We shall assume, for the ease of description in this extended abstract, that the source has no "holes", i.e., there are no three integers $i < j < \ell$ such that $n_i \neq 0, n_\ell \neq 0$, but $n_j = 0$. This is true for many, but not all, real-life distributions.

One of the properties of canonical codes is that the set of codewords having the same length are the binary representations of consecutive integers. For example, in our case, the codewords of length 9 bits are the binary integers in the range from 110011100 to 111011010. This fact can be exploited to enable efficient decoding with relatively small overhead: once a codeword of ℓ bits is detected, one can get its relative index within the sequence of codewords of length ℓ by simple subtraction.

The following information is thus needed: let $m = \min\{i \mid n_i > 0\}$ be the length of the shortest codeword, and let $base(i)$ be the integer value of the first codeword of length i. We then have

$$\begin{aligned} base(m) &= 0 \\ base(i) &= 2\,(base(i-1) \;+\; n_{i-1}) \qquad\qquad \text{for } m < i \leq k. \end{aligned}$$

Let $B_s(k)$ denote the standard s-bit binary representation of the integer k (with leading zeros, if necessary). Then the j-th codeword of length i, for $j = 0, 1, \ldots, n_i - 1$, is $B_i(base(i) + j)$. Let $seq(i)$ be the sequential index of the first codeword of length i:

$$\begin{aligned} seq(m) &= 0 \\ seq(i) &= seq(i-1) \;+\; n_{i-1} \qquad\qquad \text{for } m < i \leq k. \end{aligned}$$

Suppose now that we have detected a codeword w of length ℓ. If $I(w)$ is the integer value of the binary string w (i.e., $w = B_\ell(I(w))$), then $I(w) - base(\ell)$ is the relative index of w within the block of codewords of length ℓ. Thus $seq(\ell) + I(w) - base(\ell)$ is the relative index of w within the full list of codewords. This can be rewritten as $I(w) - diff(\ell)$, for $diff(\ell) = base(\ell) - seq(\ell)$. Thus all one needs is the list of integers $diff(\ell)$. Table 2.12 gives the values of n_i, $base(i)$, $seq(i)$ and $diff(i)$ for our example.

TABLE 2.12: *Decode values for canonical Huffman code for Zipf-200*

i	n_i	$base(i)$	$seq(i)$	$diff(i)$
3	1	0	0	0
4	3	2	1	1
5	4	10	4	6
6	8	28	8	20
7	15	72	16	56
8	32	174	31	143
9	63	412	63	349
10	74	950	126	824

We suggest in the next section a new representation of canonical Huffman codes, which not only is space-efficient, but may also speed up the decoding process, by permitting, at times, the decoding of more than a single bit in one iteration. Similar ideas, based on tables rather than on trees, were recently suggested in [26].

2.3.2 Skeleton trees for fast decoding

The following small example, using the data above, shows how such savings are possible. Suppose that while decoding, we detect that the next codeword starts with 1101. This information should be enough to decide that the following codeword ought to be of length 9 bits.

We should thus be able, after having detected the first 4 bits of this codeword, to read the following 5 bits as a block, without having to check after each bit if the end of a codeword has been reached. Our goal is to construct an efficient data-structure, that permits similar decisions as soon as they are possible. The fourth bit was the earliest possible in the above example, since there are also codewords of length 8 starting with 110.

Decoding with sk-trees

The suggested solution is a binary tree, called below an *sk-tree* (for skeleton-tree), the structure of which is induced by the underlying Huffman tree, but which has generally significantly fewer nodes. The tree will be traversed like a regular Huffman tree. That is, we start with a pointer to the root of the tree, and another pointer to the first bit of the encoded binary sequence. This sequence is scanned, and after having read a zero (resp., a 1), we proceed to the left (resp., right) son of the current node. In a regular Huffman tree, the leaves correspond to full codewords that have been scanned, so the decoding algorithm just outputs the corresponding item, resets the tree-pointer to the root and proceeds with scanning the binary string. In our case, however, we visit the tree only up to the depth necessary to identify the length of the current codeword. The leaves of the sk-tree then contain the lengths of the corresponding codewords.

```
{
    tree_pointer  $\longleftarrow$  root
    $i$  $\longleftarrow$  1
    start  $\longleftarrow$  1
    while $i <$ length_of_string
    {
        if string[$i$] $= 0$        tree_pointer  $\longleftarrow$  left(tree_pointer)
        else                     tree_pointer  $\longleftarrow$  right(tree_pointer)
        if value(tree_pointer)$> 0$
        {
            codeword  $\longleftarrow$  string[start $\cdots$ (start $+$ value(tree_pointer) $-1$)]
            output  $\longleftarrow$  table[ $I$(codeword)$-$$diff$[ value(tree_pointer)] ]
            tree_pointer  $\longleftarrow$  root
            start  $\longleftarrow$  start $+$ value(tree_pointer)
            $i$  $\longleftarrow$  start
        }
        else                     $i$  $\longleftarrow$  $i+1$
    }
}
```

FIGURE 2.13: *Decoding procedure using sk-tree*

The formal decoding process using an sk-tree is depicted in Figure 2.13. The variable *start* points to the index of the bit at the beginning of the current codeword in the encoded string, which is stored in the vector *string*[ ]. Each node of the sk-tree consists of three fields: a *left* and a *right* pointer, which are not null if the node is not a leaf, and a *value*-field, which is zero for internal nodes, but contains the length in bits of the current codeword, if the node is a leaf. In an actual implementation, we can use the fact that any internal node has either zero or two sons, and store the *value*-field and the *right*-field in the same space, with *left* = *null* serving as flag for the use of the *right* pointer. The procedure also uses two tables: *table* $[j]$, $0 \leq j < n$, giving the j-th element (in non-increasing order of frequency) of the encoded alphabet; and *diff* $[i]$ defined above, for i varying from m to k, that is from the length of the shortest to the length of the longest codeword.

The procedure passes from one level in the tree to the one below according to the bits of the encoded string. Once a leaf is reached, the next *codeword* can be read in one operation. Note that not all the bits of the input vector are individually scanned, which yields possible time savings.

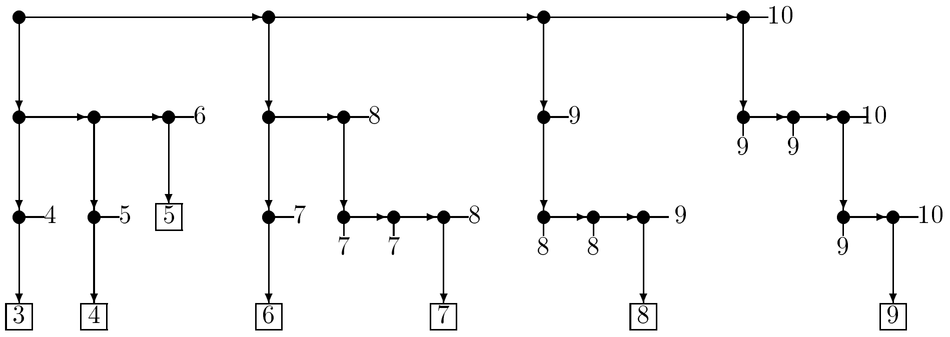


FIGURE 2.14: *sk-tree for Zipf-200 distribution*

Figure 2.14 shows the sk-tree corresponding to Zipf's distribution for $n = 200$. The tree is tilted by 45°, so that left (right) sons are indicated by arrows pointing down (to the right). The framed leaves correspond to the last codewords of the indicated length. The sk-tree of our example consists of only 49 nodes, as opposed to 399 nodes of the original Huffman tree.

Construction of sk-trees

While traversing a standard canonical Huffman tree to decode a given codeword, one may stop as soon as one gets to the root of any full subtree of depth h, for $h \geq 1$, i.e., a subtree of depth h that has 2^h leaves, since at this stage it is known that exactly h more bits are needed to complete the codeword. One way to look at sk-trees is therefore as standard Huffman trees from which all full subtrees of depth $h \geq 1$ have been pruned. A more direct and much more efficient construction is as follows.

The one-to-one correspondence between the codewords and the paths from the root to the leaves in a Huffman tree can be extended to define, for any binary string $S = s_1 \cdots s_e$, the path $P(S)$ induced by it in a tree with given root r_0. This path will consist of $e+1$ nodes r_i, $0 \leq i \leq e$, where for $i > 0$, r_i is the left (resp. right) son of r_{i-1}, if $s_i = 0$ (resp. if $s_i = 1$). For example, in Figure 2.14, $P(111)$ consists of the four nodes represented as bullets in the top line. The skeleton of the sk-tree will consist of the paths corresponding to the last codeword of every length. Let these codewords be denoted by L_i, $m \leq i \leq k$; they are, for our example, 000, 0100, 01101, 100011, etc. The idea is that $P(L_i)$ serves as “demarcation line”: any node to the left (resp. right) of $P(L_i)$, i.e., a left (resp. right) son of one of the nodes in $P(L_i)$, corresponds to a prefix of a codeword with length $\leq i$ (resp. $> i$).

As a first approximation, the construction procedure thus takes the tree obtained by $\bigcup_{i=m}^{k-1} P(L_i)$ (there is clearly no need to include the longest codeword L_k, which is always a string of k 1's), and adjoins the missing sons to turn it into a complete tree in which each internal node has both a left and a right son. The label on such a new leaf is set equal to the label of the closest leaf following it in an in-order traversal. In other words, when creating the path for L_i, one first follows a few nodes in the already existing tree, then one branches off creating new nodes; as to the labeling, the missing right son of any node in the path will be labeled $i+1$ (basing ourselves on the assumption that there are no holes), but only the missing left sons of any *new* node in the path will be labeled i.

A closer look then implies the following refinement. Suppose a codeword L_i has a zero in its rightmost position, i.e., $L_i = \alpha 0$ for some string α of length $i-1$. Then the first codeword of length $i+1$ is $\alpha 10$. It follows that only when getting to the i-th bit one can decide if the length of the current codeword is i or $i+1$. But if L_i terminates in a string of 1's, $L_i = \beta 01^a$, with $a > 0$ and $|\beta| + a = i - 1$, then the first codeword of length $i+1$ is $\beta 10^{a+1}$, so the length of the codeword can be deduced already after having read the bit following β. It follows that one does not always need the full string L_i in the sk-tree, but only its prefix up to and not including the rightmost zero. Let $L_i^* = \beta$ denote this prefix. The revised version of the above procedure starts with the tree obtained by $\bigcup_{i=m}^{k-1} P(L_i^*)$. The nodes of this tree are depicted as bullets in Figure 2.14. For each path $P(L_i^*)$ there is a leaf in the tree, and the left son of this leaf is the new terminal node, represented in Figure 2.14 by a box containing the number i. The additional leaves are then filled in as explained above.

Space complexity

To evaluate the size of the sk-tree, we count the number of nodes added by path $P(L_i^*)$, for $m \leq i < k$. Since the codewords in a canonical code, when ordered by their corresponding frequencies, are also alphabetically sorted, it suffices to compare L_i to L_{i-1}. Let $\gamma(m) = 0$, and for $i > m$, let $\gamma(i)$ be the longest common prefix of L_i and L_{i-1}, e.g., $\gamma(7)$ is the string 10 in our example. Then the number of nodes in the sk-tree is given by:

$$size = 2\left(\sum_{i=m}^{k-1} \max(0, |L_i^*| - |\gamma(i)|)\right) - 1,$$

since the summation alone is the number of internal nodes (the bullets in Figure 2.14).

The maximum function comes to prevent an extreme case in which the difference might be negative. For example, if $L_6 = 010001$ and $L_7 = 0101111$, the the longest common prefix is $\gamma(7) = 010$, but since we consider only the bits up to and not including the rightmost zero, we have $L_7^* = 01$. In this case, indeed, no new nodes are added for $P(L_7^*)$.

An immediate bound on the number of nodes in the sk-tree is $O(\min(n, k^2))$, since on the one hand, there are up to $k-1$ paths $P(L_i^*)$ of lengths $\leq k-2$, but on the other hand, it cannot exceed the number of nodes in the underlying Huffman tree, which is $2n-1$. To get a tighter bound, consider the nodes in the upper levels of the sk-tree belonging to the full binary tree F with $k-1$ leaves and having the same root as the sk-tree. The depth of F is $d = \lceil \log_2(k-1) \rceil$, and all its leaves are at level d or $d-1$. The tree F is the part of the sk-tree where some of the paths $P(L_i^*)$ must be overlapping, so we account for the nodes in F and for those below separately. There are at most $2k-1$ nodes in F; there are at most $k-1$ disjoint paths below it, with path $P(L_i^*)$ extending at most $i - 2 - \lfloor \log_2(k-1) \rfloor$ nodes below F, for $\log_2(k-1) < i \leq k$. This yields as bound for the number of nodes in the sk-tree:

$$2k + 2 \left(\sum_{i=1}^{k-2-\lfloor \log_2(k-1) \rfloor} i \right) = 2k + (k - 2 - \lfloor \log_2(k-1) \rfloor)(k - 1 - \lfloor \log_2(k-1) \rfloor).$$

There are no savings in the worst case, e.g., when there is only one codeword of each length (except for the longest, for which there are always at least two). More generally, if the depth of the Huffman tree is $\Omega(n)$, the savings might not be significant. But such trees are optimal only for some very skewed distributions. In many applications, like for most distributions of characters or character pairs or words in most natural languages, the depth of the Huffman tree is $O(\log n)$, and for large n, even the constant c, if the depth is $c \log_2 n$, must be quite small. For suppose the Huffman tree has a leaf on depth d. Then by [30, Theorem 1], the probability of the element corresponding to this leaf is $p < 1/F_{d+1}$, where F_j is the j-th Fibonacci number, and we get from [17, Exercise 1.2.1–4], that $p < (1/\phi)^{d-1}$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. Thus if $d > c \log_2 n$, we have

$$p < \left(\frac{1}{\phi} \right)^{c \log_2 n} = n^{-c \log_2(1/\phi)} = n^{-0.693c}.$$

To give a numeric example, a Huffman tree corresponding to the different words in English, as extracted from 500 MB (87 million words) of the *Wall Street Journal* [31], had $n = 289,101$ leaves. The probability for a tree of this size to have a leaf at level $3 \log_2 n$ is less than 4.4×10^{-12}, which means that even if the word with this probability appears only once, the text must be at least 4400 billion words long, enough to fill about 35,000 CD-Roms! But even if the original Huffman tree would be deeper, it is sometimes convenient to impose an upper limit of $B = O(\log n)$ on the depth, which often implies only a negligible loss in compression efficiency [19]. In any case, given a logarithmic bound on the depth, the size of the sk-tree is about

$$\log n \; (\log n - \log \log n).$$

2.4 Arithmetic Coding

We have dealt so far only with Huffman coding, and even shown that they are optimal under certain constraints. However, this optimality has often been overemphasized in the past and it is not always mentioned that Huffman codes have been shown to be optimal only for *block codes*: codes in which each new character is encoded by a fixed bit pattern made up of an integral number of bits.

The constraint of the integral number of bits had probably been considered as obvious, since the possibility of coding elements in fractional bits is quite surprising. *Arithmetic codes* overcome the limitations of block codes. In fact, arithmetic codes have had a long history [32, 33], but became especially popular after Witten, Neal and Cleary's paper [34] in 1987.

The approach taken by arithmetic coding is quite different from that of Huffman coding. Instead of using the probabilities of the different characters to generate codewords, it defines a *process* in the course of which a binary number is generated. Each new character of the text to be encoded allows a more precise determination of the number. When the last character is processed, the number is stored or transmitted.

The encoding process starts with the interval $[0, 1)$, which will be narrowed repeatedly. We assign to each character a sub-interval, the size of which is proportional to the probability of occurrence of the character. Processing a certain character x is then performed by replacing the current interval by the sub-interval corresponding to x. Refer to the example in Figure 2.15. We assume our alphabet consists of the four characters {A, B, C, D}, appearing with probabilities 0.4, 0.3, 0.1 and 0.2, respectively. We arbitrarily choose a corresponding partition of the interval $[0, 1)$, for example, $[0, 0.1)$ for C, $[0.1, 0.4)$ for B, $[0.4, 0.8)$ for A and finally $[0.8, 1)$ for D. This partition is depicted as the leftmost bar in Figure 2.15.

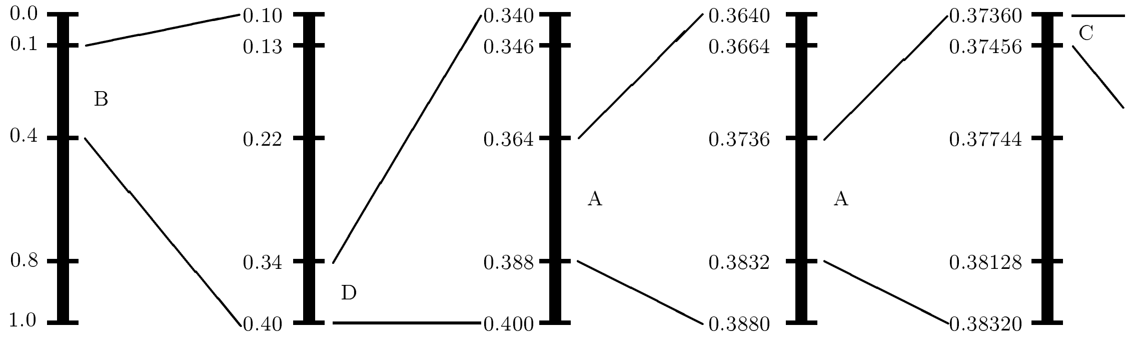


FIGURE 2.15: *Example of arithmetic coding*

Suppose now that the text we wish to encode is BDAAC. The first character is B, so the new interval after the encoding of B is $[0.1, 0.4)$. This interval is now partitioned similarly to the original one, i.e., the first 10% are assigned to C, the next 30% to B, etc. The new sub-division can be seen next to the second bar from the left. The second character to be encoded is D, so the corresponding interval is $[0.34, 0.40)$. Repeating now the process, we see that the next character, A, narrows the chosen sub-interval further to $[0.364, 0.388)$, and the next A to $[0.3736, 0.3832)$, and finally the last C to $[0.37360, 0.37456)$.

To allow unambiguous decoding, it is this last interval that should be transmitted. This would, however, be rather wasteful: as more characters are encoded, the interval will get narrower, and many of the leftmost digits of its upper limit will overlap with those of its lower limit. In our example, both limits start with 0.37. One can overcome this inefficiency and transmit only a single number if some additional information is given. For instance, if the number of characters is also given to the decoder, or, as is customary, a special end-of-file character is added at the end of the message, it suffices to transmit any single number within the final interval. In our case, the best choice would be $y = 0.3740234375$, because its binary representation 0.0101111111 is the shortest among the numbers of the interval.

Decoding is then just the inverse of the above process. Since y is between 0.1 and 0.4, we know that the first character must be `B`. If so, the interval has been narrowed to $[0.1, 0.4)$. We thus seek the next sub-interval which contains y, and find it to be $[0.34, 0.40)$, which corresponds to `D`, etc. Once we get to $[0.37360, 0.37456)$, the process has to be stopped by some external condition, otherwise we could continue this decoding process indefinitely, for example by noting that y belongs to $[0.373984, 0.374368)$, which could be interpreted as if the following character were `A`, etc.

As has been mentioned, the longer the input string, the more digits or bits are needed to specify a number encoding the string. Compression is achieved by the fact that a frequently occurring character only slightly narrows the current interval. The number of bits needed to represent a number depends on the required precision. The smaller the given interval, the higher precision is necessary to specify a number in it; if the interval size is p, $\lceil -\log_2 p \rceil$ bits might be needed.

To evaluate the number of bits necessary by arithmetic coding, we recall the notation used in Section 2.1. The text consists of characters $x_1 x_2 \cdots x_W$, each of which belongs to an alphabet $\{a_1, \ldots, a_n\}$. Let w_i be the number of occurrences of letter a_i, so that $W = \sum_{i=1}^{n} w_i$ is the total length of the text, and let $p_i = w_i/W$ be the probability of occurrence of letter a_i, $1 \leq i \leq n$. Denote by p_{x_j} the probability associated with the j-th character of the text.

After having processed the first character, x_1, the interval has been narrowed to size p_{x_1}, after the second character, the interval size is $p_{x_1} p_{x_2}$, etc. We get that the size of the final interval after the whole text has been processed is $p_{x_1} p_{x_2} \cdots p_{x_W}$. Therefore the number of bits needed to encode the full text is

$$\begin{aligned} -\log_2 \left(\prod_{j=1}^{W} p_{x_j} \right) &= -\sum_{j=1}^{W} \log_2 p_{x_j} = -\sum_{i=1}^{n} w_i \log_2 p_i \\ &= W \left(-\sum_{i=1}^{n} p_i \log_2 p_i \right) = W\ H, \end{aligned}$$

where we get the second equality by summing over the letters of the alphabet with their frequency instead of summing over the characters of the text, and where H is the entropy of the given probability distribution. Amortizing this per character, we get that the average number of bits needed to encode a single character is just H, which has been shown in eqn. (2.4) to be the information theoretic lower bound.

We conclude that from the point of view of compression, arithmetic coding has an optimal performance. But our presentation and the analysis are oversimplified: they do not take into account the overhead incurred by the end_of_file character, nor the fractions of bits lost by alignment for each block to be encoded. It can be shown [28] that although these additions are often negligible relative to the average size of a codeword, they might be significant relative to the *difference* between the codeword lengths for Huffman and arithmetic codes. There are also other technical problems, such as the limited precision of our computers, which does not allow the computation of a single number for a long text; there is thus a need for incremental transmission, which further complicates the algorithms, see [34].

In spite of the optimality of arithmetic codes, Huffman codes may still be the preferred choice in many applications: they are much faster for encoding and especially decoding, they are less error prone, and after all, the loss in compression efficiency, if any, is generally very small.

2.5 Dictionary Based Text Compression

The text compression methods we have seen so far are called *statistical* methods, as they exploit the skewness of the distribution of occurrence of the characters. Another family of compression methods is based on *dictionaries*, which replace variable length substrings of the text by (shorter) pointers to a dictionary in which a collection of such substrings has been stored. Depending on the application and the implementation details, each method can outperform the other.

Given a fixed amount of RAM which we would allocate for the storage of a dictionary, the selection of an optimal set of strings to be stored in the dictionary turns out to be a difficult task, because the potential strings are overlapping. A similar problem is shown to be NP-complete in [35], but more restricted versions of this problem of optimal dictionary construction are tractable [36].

For IR applications, the dictionary ought to be fixed, since the compressed text need be accessed randomly. For the sake of completeness, however, we mention also *adaptive* techniques, which are the basis of most popular compression methods. Many of these are based on two algorithms designed by Lempel and Ziv [37, 38].

In one of the variants of the first algorithm [37], often referred to as LZ77, the dictionary is in fact the previously scanned text, and pointers to it are of the form (d, ℓ), where d is an offset (the number of characters from the current location to the previous occurrence of a substring matching the one that starts at the current location), and ℓ is the length of the matching string. There is therefore no need to store an explicit dictionary. In the second algorithm [38], the dictionary is dynamically expanded by adjoining sub-strings of the text that could not be parsed. For more details on LZ methods and their variants, the reader is referred to [25].

Even once the dictionary is given, the compression scheme is not yet well defined, as one must decide how to *parse* the text into a sequence of dictionary elements. Generally, the parsing is done by a *greedy* method, i.e., at any stage, the longest matching element from the dictionary is sought. A greedy approach is fast, but not necessarily optimal. Because the elements of the dictionary are often overlapping, and particularly for LZ77 variants, where the dictionary is the text itself, a different way of parsing might yield better compression. For example, assume the dictionary consists of the strings $D = \{$`abc`, `ab`, `cdef`, `d`, `de`, `ef`, `f`$\}$ and that the text is $S =$ `abcdef`; assume further that the elements of D are encoded by some fixed-length code, which means that $\lceil \log_2(|D|) \rceil$ bits are used to refer to any of the elements of D; then parsing S by a greedy method, trying to match always the longest available string, would yield `abc-de-f`, requiring three codewords, whereas a better partition would be `ab-cdef`, requiring only two.

The various dictionary compression methods differ also by the way they encode the elements. This is most simply done by a fixed length code, as in the above example. Obviously, different encoding methods might yield different optimal parsings. Returning to the above example, if the elements `abc`, `d`, `de`, `ef`, `f`, `ab`, `cdef` of D are encoded respectively by 1, 2, 3, 4, 5, 6 and 6 bits, then the parsing `abc-de-f` would need nine bits for its encoding, and for the encoding of the parsing `ab-cdef`, 12 bits would be needed. The best parsing, however, for the given codeword lengths, is `abc-d-ef`, which is neither a greedy parsing, nor does it minimize the number of codewords, and requires only seven bits.

The way to search for the optimal parsing is by reduction to a well-known graph theoretical problem. Consider a text string S consisting of a sequence of n characters $S_1S_2\cdots S_n$, each character S_i belonging to a fixed alphabet Σ. Substrings of S are referenced by their limiting indices, i.e., $S_i\cdots S_j$ is the substring starting at the i-th character in S, up to and including the j-th character. We wish to compress S by means of a dictionary D, which is a set of character strings $\{\sigma_1, \sigma_2, \ldots\}$, with $\sigma_i \in \Sigma^+$. The dictionary may be explicitly given and finite, as in the example above, or it may be potentially infinite, e.g., for the Lempel-Ziv variants, where any previously occurring string can be referenced.

The compression process consists of two independent phases: parsing and encoding. In the *parsing* phase, the string S is broken into a sequence of consecutive sub-strings, each belonging to the dictionary D, i.e., an increasing sequence of indices $i_0 = 0, i_1, i_2, \ldots$ is found, such that

$$S \quad = \quad S_1S_2\cdots S_n \quad = \quad S_1\cdots S_{i_1}\, S_{i_1+1}\cdots S_{i_2}\ \cdots,$$

with $S_{i_j+1}\cdots S_{i_{j+1}} \in D$ for $j = 0, 1, \ldots$. One way to assure that at least one such parsing exists is to force the dictionary D to include each of the individual characters of Σ. The second phase is based on an *encoding* function $\lambda : D \longrightarrow \{0,1\}^*$, that assigns to each element of the dictionary a binary string, called its encoding. The assumption on λ is that it produces a code which is UD. This is most easily obtained by a fixed length code, but as has been seen earlier, a sufficient condition for a code being UD is to choose it as a prefix code.

The problem is the following: given the dictionary D and the encoding function λ, we are looking for the optimal partition of the text string S, i.e., the sequence of indices $i_1, i_2, \ldots$ is sought, that minimizes $\sum_{j\geq 0} |\lambda(S_{i_j+1}\cdots S_{i_{j+1}})|$.

To solve the problem, a directed, labeled graph $G = (V, E)$ is defined for the given text S. The set of vertices is $V = \{1, 2, \ldots, n, n+1\}$, with vertex i corresponding to the character S_i for $i \leq n$, and $n+1$ corresponding to the end of the text; E is the set of directed edges: an ordered pair (i, j), with $i < j$, belongs to E if and only if the corresponding substring of the text, that is, the sequence of characters $S_i\cdots S_{j-1}$, can be encoded as a single unit. In other words, the sequence $S_i\cdots S_{j-1}$ must be a member of the dictionary, or more specifically for LZ77, if $j > i+1$, the string $S_i\cdots S_{j-1}$ must have appeared earlier in the text. The label L_{ij} is defined for every edge $(i, j) \in E$ as $|\lambda(S_i\cdots S_{j-1})|$, the number of bits necessary to encode the corresponding member of the dictionary, for the given encoding scheme at hand. The problem of finding the optimal parsing of the text, relative to the given dictionary and the given encoding scheme, therefore reduces to the well-known problem of finding the shortest path in G from vertex 1 to vertex $n+1$. In our case, there is no need to use Dijkstra's algorithm, since the directed graph contains no cycles, all edges being of the form (i, j) with $i < j$. Thus by a simple dynamic programming method, the shortest path can be found in time $O(|E|)$.

Figure 2.16 displays a small example of a graph, corresponding to the text `abbaabbabab`

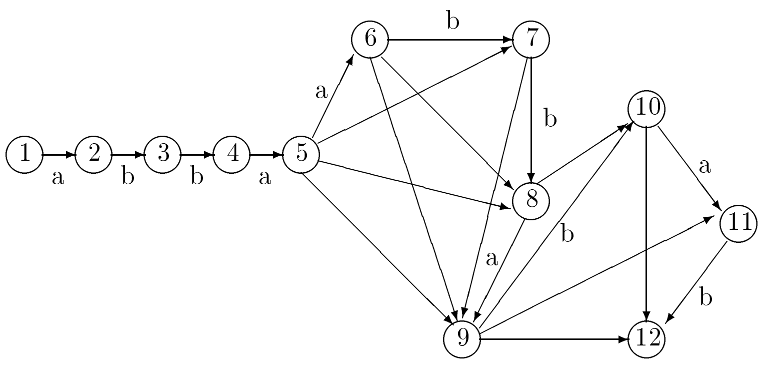


FIGURE 2.16: Graph corresponding to text `abbaabbabab`

and assuming that LZ77 is used. The edges connecting vertices i to $i+1$, for $i = 1, \dots, n$, are labeled by the character S_i.

As an example of an encoding scheme, we refer to the on-the-fly compression routine recently included in a popular operating system. It is based on [39], a variant of LZ77, using hashing on character pairs to locate (the beginning of) recurrent strings. The output of the compression process is thus a sequence of elements, each being either a single (uncompressed) character, or an offset-length pair (d, ℓ). The elements are identified by a flag bit, so that a single character is encoded by a zero, followed by the 8-bit ASCII representation of the character, and the encoding of each (d, ℓ) pair starts with a 1. The sets of possible offsets and lengths are split into classes as follows: let $B_m(n)$ denote the standard m-bit binary representation of n (with leading zeros if necessary), then, denoting the encoding scheme by λ_M:

$$\lambda_M(\text{offset } d) = \begin{cases} 1B_6(d-1) & \text{if } 1 \le d \le 64 \\ 01B_8(d-65) & \text{if } 64 < d \le 320 \\ 11B_{12}(d-321) & \text{if } 320 < d \le 4416 \end{cases}$$

$$\lambda_M(\text{length } \ell) = \begin{cases} 0 & \text{if } \ell = 2 \\ 1^{j+1} \;\; 0 \;\; B_j(\ell - 2 - 2^j) & \text{if } 2^j \le \ell - 2 < 2^{j+1}, \qquad \text{for } j = 0, 1, 2, \dots \end{cases}$$

For example, the first few length encodings are: 0, 10, 1100, 1101, 111000, 111001, 111010, 111011, 11110000, etc. Offsets are thus encoded by 8, 11 or 15 bits, and the number of bits used to encode the lengths ℓ is 1 for $\ell = 2$ and $2\lceil \log_2(\ell - 1) \rceil$ for $\ell > 2$.

3. DICTIONARIES

All large full-text retrieval systems make extensive use of dictionaries of all kinds. They are needed to quickly access the concordance, they may be used for compressing the text itself and they generally provide some useful additional information which can guide the user in the choice of his keywords.

Dictionaries can of course be compressed as if they were regular text, but taking their special structure into account may lead to improved methods [40]. A simple, yet efficient, technique is the *Prefix Omission Method* (POM), a formal definition of which can be found in [2], where it is called *front-end compression*.

The method is based on the observation that consecutive entries in a dictionary mostly share some leading letters. Let x and y be consecutive dictionary entries and let m be the length (number of letters) of their longest common prefix. Then it suffices to store this common prefix only once (with x) and to omit it from the following entry, where instead the length m will be kept. This is easily generalized to a longer list of dictionary entries, as in the example in Figure 3.1:

dictionary entry	*prefix length*	*stored suffix*
FORM	0	FORM
FORMALLY	4	ALLY
FORMAT	5	T
FORMATION	6	ION
FORMULATE	4	ULATE
FORMULATING	8	ING
FORTY	3	TY
FORTHWITH	4	HWITH

FIGURE 3.1: *Example of the Prefix Omission Method*

Note that the value given for the prefix length does not refer to the string which was actually stored, but rather to the corresponding full-length dictionary entry. The compression and decompression algorithms are immediate.

If the dictionary entries are coded in standard format, with one byte per character, one could use the first byte of each entry in the compressed dictionary to store the value of m. There will mostly be a considerable gain, since the average length of common prefixes of consecutive entries in large dictionaries is generally much larger than 1. Even when the entries are already compressed, for example by a character by character Huffman code, one would still achieve some savings. For convenience, one could choose a fixed integer parameter k and reserve the first k bits of every entry to represent values of m for $0 \leq m < 2^k$, where k is not necessarily large enough to accommodate the longest omitted prefix. In the above example, k could for example be chosen as 3, and the entry corresponding to FORMULATING would then be $(7, \texttt{TING})$.

A standard dictionary does however not provide the flexibility required by sophisticated systems. For instance, a prominent feature would be the possibility of processing truncated terms of several kinds by means of a variable-length don't-care character $*$. Examples of the use of $*$ for prefix, suffix and infix truncation have been given in Section 1.

Suffix truncation can be handled by the regular dictionary. To enable prefix truncation, the problem is that the relevant terms are scattered throughout the file and therefore hard to locate. A possible solution is to adjoin an *inverse* dictionary to the system: for each term, form its reversed string, then sort the reversed strings lexicographically. To search, e.g., for $*$`ache`, we would access the inverse dictionary with the string `ehca`, retrieve the entries prefixed by it (they form a contiguous block), e.g., `ehcadaeh` and `ehcahtoot`, and reverse these strings again to get our terms, e.g., `headache` and `toothache`. The solution of the inverse dictionary can not be extended to deal with prefix and suffix truncation simultaneously.

An elegant method allowing the processing of any kind of truncation is the *permuted dictionary* suggested in [2]. Given a dictionary, the corresponding permuted dictionary is obtained by the following sequence of steps:

1. append to each term a character `/` which does not appear in any term;
2. for a term x of length n characters, form $n+1$ new terms by cyclically shifting the string x`/` by k characters, $0 \leq k \leq n$;
3. sort the resulting list alphabetically.

Figure 3.2 shows these steps for the dictionary consisting of the strings `JACM`, `JASIS` and `IPM`. The first column lists the terms with the appended `/`. In the second column, the permuted terms generated by the same original term appear consecutively, and the third column is sorted. The last column shows how the permuted dictionary can be compressed by POM.

original	*permuted*	*sorted*	compressed	
			m	*suffix*
JACM	JACM/	/IPM	0	/IPM
JASIS	ACM/J	/JACM	1	JACM
IPM	CM/JA	/JASIS	3	SIS
	M/JAC	ACM/J	0	ACM/J
	/JACM	ASIS/J	1	SIS/J
	JASIS/	CM/JA	0	CM/JA
	ASIS/J	IPM/	0	IPM/
	SIS/JA	IS/JAS	1	S/JAS
	IS/JAS	JACM/	0	JACM/
	S/JASI	JASIS/	2	SIS/
	/JASIS	M/IP	0	M/IP
	IPM/	M/JAC	2	JAC
	PM/I	PM/I	0	PM/I
	M/IP	S/JASI	0	S/JASI
	/IPM	SIS/JA	1	IS/JA

FIGURE 3.2: *Example of the permuted dictionary*

The key for using the permuted dictionary efficiently is a function get(x), which accesses the file and retrieves all the strings having x as prefix. These strings are easily located since they appear consecutively, and the corresponding original terms are recovered by a simple cyclic shift. To process truncated terms, all one needs is to call get() with the appropriate parameter. Figure 3.3 shows in its leftmost columns how to deal with suffix, prefix, infix, and simultaneous prefix and suffix truncations. The other columns then bring an example for each of these categories: first the query itself, then the corresponding call to `get()`, the retrieved entries from the permuted dictionary, and the corresponding reconstructed terms.

`X*`	`get(/X)`	`JA*`	`get(/JA)`	`/JACM, /JASIS`	`JACM, JASIS`
`*X`	`get(X/)`	`*M`	`get(M/)`	`M/IP, M/JAC`	`IPM, JACM`
`X*Y`	`get(Y/X)`	`J*S`	`get(S/J)`	`S/JASI`	`JASIS`
`*X*`	`get(X)`	`*A*`	`get(A)`	`ACM/J, ASIS/J`	`JACM, JASIS`

FIGURE 3.3: *Processing truncated terms with permuted dictionary*

4. CONCORDANCES

Every occurrence of every word in the database can be uniquely characterized by a sequence of numbers that give its exact position in the text. Typically, such a sequence would consist of the document number d, the paragraph number p (in the document), the sentence number s (in the paragraph) and the word number w (in the sentence). The quadruple (d, p, s, w) is the *coordinate* of the occurrence, and the corresponding fields will be called for short d-field, p-field, s-field and w-field. In the sequel, we assume for the ease of discussion that coordinates of every retrieval system are of this form; however, all the methods can also be applied to systems with different coordinate structure, such as book-page-line-word, etc. The concordance contains, for every word of the dictionary, the lexicographically ordered list of all its coordinates in the text; it is accessed via the dictionary that contains for every word a pointer to the corresponding list in the concordance. The concordance is kept in compressed form on secondary storage and parts of it are fetched when needed and decompressed. The compressed file is partitioned into equi-sized blocks such that one block can be read by a single I/O operation.

Since the list of coordinates of any given word is ordered, adjacent coordinates will often have the same d-field, or even the same d- and p-fields, and sometimes, especially for high frequency words, identical d-, p- and s-fields. Thus POM can be adapted to the compression of concordances, where to each coordinate a *header* is adjoined, giving the *number of fields* which can be copied from the preceding coordinate; these fields are then omitted. For instance in our model with coordinates (d, p, s, w), it would suffice to keep a header of 2 bits. The four possibilities are: don't copy any field from the previous coordinate, copy the d-field, copy d- and p-field and copy d-, p- and s-field. Obviously, different coordinates cannot have all four fields identical.

For convenient computer manipulation, one generally chooses a fixed length for each field, which therefore has to be large enough to represent the maximal possible values. However, most stored values are small, thus there is usually much wasted space in each coordinate. In some situations, some space can be saved at the expense of a longer processing time, as in the following example.

At RRP, the maximal length of a sentence is 676 words! Such long sentences can be explained by the fact that in the Responsa literature punctuation marks are often omitted or used very scarcely. At TLF, there is even a “sentence” of more than 2000 words (a modern poem). Since on the other hand most sentences are short and it was preferred to use only field-sizes which are multiples of half-bytes, the following method is used: the size of the w-field is chosen to be one byte (8 bits); any sentence of length $\ell > 256$ words, such that $\ell = 80k + r$ $(0 \leq r < 80)$, is split into k units of 80 words, followed (if $r > 0$) by a sentence of r words. These sentences form only a negligible percentage of the database. While resolving the storage problem, the insertion of such “virtual points” in the middle of a sentence creates some problems for the retrieval process. When in a query one asks to retrieve occurrences of keywords A and B such that A and B are adjacent or that no more than some small number of words appear between them, one usually does not allow A and B to appear in different sentences. This is justified, since “adjacency” and “near vicinity” operators are generally used to retrieve expressions, and not the coincidental juxtaposition of A at the end of a sentence with B at the beginning of the following one. However in the presence of virtual points, the search should be extended also into neighboring “sentences”, if necessary, since the virtual points are only artificial boundaries which might have split some interesting expression. Hence this solution further complicates the retrieval algorithms.

The methods presented in the next section not only yield improved compression, but also get rid of the virtual points.

4.1 Using Variable-Length Fields

The basic idea of all the new methods is to allow the p-, s- and w-fields to have variable length. As in POM, each compressed coordinate will be prefixed by a header which will encode the information necessary to decompress the coordinate. The methods differ in their interpretation of the header. The choice of the length of every field is based on statistics gathered from the entire database on the distribution of the values in each field. Thus for dynamically changing databases, the compression method would need frequent updates, so that the methods are more suitable for retrieval systems with static databases. However, if the text changes only slowly, say it is a large corpus to which from time to time some documents are adjoined which have characteristics similar to the documents already in the corpus, then the methods will still perform well, though not optimally.

The codes in the header can have various interpretations: they can stand for a length ℓ, indicating that the corresponding field is encoded in ℓ bits; they can stand for a certain value v, indicating that the corresponding field contains that value; they can finally indicate that no value for the corresponding field is stored and that the value of the preceding coordinate should be used. This is more general than the prefix-omission technique, since one can decide for every field individually whether or not to omit it, while in POM, the p-field is only omitted if the d-field is, etc.

The d-field is treated somewhat differently. Since this is the highest level of the hierarchy in our model, this field may contain also very large numbers (there are rarely 500 words in a sentence or 500 sentences in a paragraph, but a corpus may contain tens of thousands of documents). Moreover, the d-fields of most coordinates will contain values, in the representation of which one can save at most one or two bits, if at all. On the other hand, the d-field is the one where the greatest savings are achieved by POM. Thus we shall assume in the sequel that for the d-field, we just keep one bit in the header, indicating whether the value of the preceding coordinate should be copied or not; if not, the d-field will appear in its entire length.

We now describe the specific methods in detail.

A. The simple method. The header contains codes for the size (in bits) of every field.

(i) Allocate two bits for each of the p-, s- and w-fields, giving four possible choices for each.

We consider the following variations:

a. One of the possible codes indicates the omission of the field, thus we are left with only 3 possible choices for the length of each field.

b. The four choices are used to encode field-lengths, thus not allowing the use of the preceding coordinate.

c. Use a for the p- and s-fields, and b for the w-field.

Method A(i)c is justified by the fact that consecutive coordinates having the same value in their w-field are rare (3.5% of the concordance at RRP). The reason is that this corresponds to a certain word appearing in the same relative location in different sentences, which is mostly a pure coincidence; on the other hand consecutive coordinates having the same value in one of their other fields correspond to a certain word appearing more than once in the same sentence, paragraph or document, and this occurs frequently. For instance, at RRP, 23.4% of the coordinates have the same s-field as their predecessors, 41.7% have the same p-field and 51.6% have the same d-field.

Note that the header does not contain the binary encoding of the lengths, since this would require a larger number of bits. By storing a *code* for the lengths the header is kept smaller, but at the expense of increasing decompression time, since a table is needed which translates the codes into actual lengths. This remark applies also to the subsequent methods.

(ii) Allocate three bits in the header for each of the p-, s- and w-fields, giving 8 possible choices for each.

The idea of (ii) is that by increasing the number of possibilities (and hence the overhead for each coordinate), the range of possible values can be partitioned more efficiently, which should lead to savings in the remaining part of the coordinate. Again three methods corresponding to a, b and c of (i) were checked.

B. Using some fields to encode frequent values.

For some very frequent values, the code in the header will be interpreted directly as one of the values, and not as the length of the field in which they are stored. Thus the corresponding field can be omitted in all these cases. However, the savings for the frequent values come at the expense of reducing the number of possible choices for the lengths of the fields for the less frequent values. For instance, at RRP, the value 1 appears in the s-field of more than 9 million coordinates (about 24% of the concordance), thus all these coordinates will have no s-field in their compressed form, and the code in the part of the header corresponding to the s-field will be interpreted as "value 1 in the s-field".

(i) Allocate 2 bits in the header for each of the p-, s- and w-fields; one of the codes points to the most frequent value.

(ii) Allocate 3 bits in the header for each of the p-, s- and w-fields; three of the codes point to the 3 most frequent values.

There is no subdivision into methods a, b and c as in A (in fact the method used corresponds to a), because we concluded from our experiments that it is worth to keep the possibility of using the previous coordinate in case of equal values in some field. Hence one code was allocated for this purpose, which left only 2 codes to encode the field-lengths in (i) and 4 codes in (ii). For (ii) we experimented also with allowing 2 or 4 of the 8 possible choices to encode the 2 or 4 most frequent values; however, on our data, the optimum was always obtained for 3. There is some redundancy in the case of consecutive coordinates having both the same value in some field, and this value being the most frequent one. There are then two possibilities to encode the second coordinate using the same number of bits. In such a case, the code for the frequent value should be preferred over the one pointing to the previous coordinate, as decoding of the former is usually faster.

C. Combining methods A and B.

Choose individually for each of the p-, s- and w-fields, the best of the previous methods.

D. Encoding length-combinations.

If we want to push the idea of A further, we should have a code for *every* possible length of a field, but the maxima of the values can be large. For example, at RRP, one needs 10 bits for the maximal value of the w-field, 9 bits for the s-field and 10 bits for the p-field. This would imply a header length of 4 bits for each of these fields, which cannot be justified by the negligible improvement over method A(ii).

The size of the header can be reduced by replacing the three codes for the sizes of the p-, s- and w-fields by a single code in the following way. Denote by l_p, l_s and l_w the lengths of the p-, s- and w-fields respectively, i.e., the sizes (in bits) of the binary representations without leading zeros of the values stored in them. In our model $1 \leq l_p, l_s, l_w \leq 10$, so there are up to 10^3 possible triplets (l_p, l_s, l_w). However, most of these length-combinations occur only rarely, if at all. At RRP, the 255 most frequent (l_p, l_s, l_w)-triplets account already for 98.05% of the concordance. Therefore

(i) Allocate 9 bits as header, of which 1 bit is used for the d-field; 255 of the possible codes in the remaining 8 bits point to the 255 most frequent (l_p, l_s, l_w)-triplets; the last code is used to indicate that the coordinate corresponds to a "rare" triplet, in which case the p-, s- and w-fields appear already in their decompressed form.

Although the "compressed" form of the rare coordinates, including a 9-bit header, may in fact need more space than the original coordinate, we still save on the average.

Two refinements are now superimposed. We first note that one does not need to represent the integer 0 in any field. Therefore one can use a representation of the integer $n-1$ in order to encode the value n, so that only $\lfloor \log_2(n-1) \rfloor + 1$ bits are needed instead of $\lfloor \log_2 n \rfloor + 1$. This may seem negligible, because only one bit is saved and only when n is a power of 2, thus for very few values of n. However, the first few of these values, 1, 2 and 4, appear very frequently, so that in fact this yields a significant improvement. At RRP, the total size of the compressed p-, s- and w-fields (using method D) was further reduced by 7.4%, just by shifting the stored values from n to $n-1$.

The second refinement is based on the observation that since we know from the header the exact length of each field, we know the position of the left-most 1 in it, so that this 1 is also redundant. The possible values in the fields are partitioned into classes $\mathcal{C}_i$ defined by $\mathcal{C}_0 = \{0\}$, $\mathcal{C}_i = \{\ell : 2^{i-1} \le \ell < 2^i\}$, and the header gives for the values in each of the p-, s- and w-fields, the indices i of the corresponding classes. Therefore if $i \le 1$, there is no need to store any additional information because $\mathcal{C}_0$ and $\mathcal{C}_1$ are singletons, and for $\ell \in \mathcal{C}_i$ for $i > 1$, only the $i-1$ bits representing the number $\ell - 2^{i-1}$ are kept. For example, suppose the values in the p-, s- and w-fields are 3, 1 and 28. Then the encoded values are 2, 0 and 27 which belong to $\mathcal{C}_2$, $\mathcal{C}_0$ and $\mathcal{C}_5$ respectively. The header thus points to the triplet $(2, 0, 5)$ (assuming that this is one of the 255 frequent ones) and the rest of the coordinate consists of the five bits 01011, which are parsed from left to right as 1 bit for the p-field, 0 bits for the s-field and 4 bits for the w-field. A similar idea was used in [15] for encoding run-lengths in the compression of sparse bit-vectors.

(ii) Allocate 8 bits as header of which 1 bit is used for the d-field; the remaining 7 bits are used to encode the 127 most frequent (l_p, l_s, l_w)-triplets.

The 127 most frequent triplets still correspond to 85.19% of the concordance at RRP. This is therefore an attempt to save one bit in the header of each coordinate at the expense of having more non-compressed coordinates.

Another possibility is to extend method D also to the d-field. Let b be a Boolean variable corresponding to the two possibilities for the d-field, namely T = the value is identical to that of the preceding coordinate, thus omit it, or F = different value, keep it. We therefore have up to 2000 quadruples (b, l_p, l_s, l_w), which are again sorted by decreasing frequency.

(iii) Allocate 8 bits as header; 255 of the codes point to the 255 most frequent quadruples.

At RRP, these 255 most frequent quadruples cover 87.08% of the concordance. For the last two methods, one could try to get better results by compressing also some of the coordinates with the non-frequent length combinations, instead of storing them in their decompressed form. We did not, however, pursue this possibility.

Encoding

After choosing the appropriate compression method, the concordance is scanned sequentially and each coordinate is compressed with or without using the preceding one. For each of the above methods, the length of the header is constant, thus the set of compressed coordinates forms a prefix-code. Therefore the compressed coordinates, which have variable lengths, can simply be concatenated. The compressed concordance consists of the resulting very long bit-string. This string is partitioned into blocks of equal size, the size corresponding to the buffer-size of a read/write operation. If the last coordinate in a block does not fit there in its entirety, it is moved to the beginning of the next block. The first coordinate of each block is considered as having no predecessor, so that if in the original encoding process a coordinate which is the first in a block referred to the previous coordinate, this needs to be corrected. This allows now to access each block individually, while adding only a negligible number of bits to each block.

Decoding

Note that for a static information retrieval system, encoding is done only once (when building the database), whereas decoding directly affects the response-time for on-line queries. In order to increase the decoding speed, we use a small precomputed table $\mathcal{T}$ which is stored in internal memory. For a method with header length k bits, this table has 2^k entries. In entry i of $\mathcal{T}$, $0 \le i < 2^k$, we store the relevant information for the header consisting of the k-bit binary representation of the integer i.

For the methods in A, the relevant information simply consists of the lengths, P, S and W, of the p-, s- and w-fields (recall that we assume that only one bit is kept in the header for the d-field, so either the d-field appears in its entire length D, which is constant, or it is omitted), and of the sum of all these lengths (including D), which is the length of the remaining part of the coordinate. We shall use the following notations: for a given internal structure of a decompressed coordinate, let h_d, h_p, h_s and h_w be the indices of the leftmost bit of the d-, p-, s- and w-fields respectively, the index of the rightmost bit of a coordinate being 0. For example with a 4 byte coordinate and one byte for each field we would have $h_d = 31$, $h_p = 23$, $h_s = 15$ and $h_w = 7$; these values are constant for the entire database. COOR and LAST are both addresses of a contiguous space in memory in which a single decompressed coordinate can fit (hence of length $h_d + 1$ bits). The procedure SHIFT(X, y, z) shifts the substring of X which is obtained by ignoring its y rightmost bits, by z bits to the left. Then the following loop could be used for the decoding of a coordinate:

```
1.   loop while there is more input or until a certain coordinate is found
2.       $H \leftarrow$ next $k$ bits                          // read header
3.       (TOT, $P, S, W$) $\leftarrow \mathcal{T}(H)$              // decode header using table
4.       COOR $\leftarrow$ next TOT bits                  // right justified suffix of coordinate
5.       SHIFT(COOR, $W, h_w - W$)                 // move d-, p- and s-field
6.       SHIFT(COOR, $h_w + S, h_s - S$)            // move d- and p-field
7.       SHIFT(COOR, $h_s + P, h_p - P$)            // move d-field
8.       if TOT $= P + S + W$ then copy d-field from LAST
9.       if $P = 0$ then copy p-field from LAST
10.      if $S = 0$ then copy s-field from LAST
11.      if $W = 0$ then copy w-field from LAST
```

```
12.        LAST ← COOR
13.    end of loop
```

There is no need to initialize LAST, since the first coordinate of a block never refers to the preceding coordinate.

For the methods in B and C, we store sometimes actual values, and not just the lengths of the fields. This can be implemented by using negative values in the table $\mathcal{T}$. For example, if $P = -2$, this could be interpreted as "value 2 in the p-field". Note that when the value stored in a field is given by the header, this field has length 0 in the remaining part of the coordinate. Thus we need the following updates to the above algorithm: line 3 is replaced by

```
$(\mathsf{TOT}, P1, S1, W1) \leftarrow \mathcal{T}(H)$
if $P1 < 0$ then $P \leftarrow 0$ else $P \leftarrow P1$
```

and statements similar to the latter for the s- and w-fields. After statement 11 we should insert

```
if $P1 < 0$ then put $-P1$ in p-field of COOR
```

and similar statements for the s- and w-fields.

The decoding of the methods in D is equivalent to that of A. The only difference is in the preparation of the table $\mathcal{T}$ (which is done only once). While for A to each field correspond certain fixed bits of the header which determine the length of that field, for D the header is non-divisible and represents the lengths of all the fields together. This does not affect the decoding process, since in both methods a table-lookup is used to interpret the header. An example of the encoding and decoding processes appears in the next section.

Parameter Setting

All the methods of the previous section were compared on the concordance of RRP. Each coordinate had a (d, p, s, w)-structure and was of length 6 bytes (48 bits). Using POM, the average length of a compressed coordinate was 4.196 bytes, i.e., a compression gain of 30%.

Table 4.1 gives the frequencies of the first few values in each of the p-, s- and w-fields, both with and without taking into account the previous coordinate. The frequencies are given in cumulative percentages, e.g., the row entitled s-field contains in the column headed i the percentage of coordinates having a value $\leq i$ in their s-field. We have also added the values for which the cumulative percentage first exceeds 99%.

As one can see, the first four values in the p- and s-fields account already for half of the concordance. This means that most of the paragraphs consist of only a few sentences and most of the documents consist of only a few paragraphs. The figures for the w-field are different, because short sentences are not preponderant. While the (non-cumulative) frequency of the values i in the s-field is a clearly decreasing function of i, it is interesting to note the peek at value 2 for the p-field. This can be explained by the specific nature of the Responsa literature, in which most of the documents have a question-answer structure. Therefore the first paragraph of a document usually contains just a short question, whereas the answer, starting from the second paragraph, may be much longer.

Table 4.1: *Distribution of values stored in p-, s- and w-fields*

Value		1	2	3	4	5	79	83	87	93	119	120
ignoring	p-field	14.1	35.2	46.5	54.2	60.2		99				
preceding	s-field	24.2	40.2	51.1	58.8	64.5	99					
coordinate	w-field	3.0	5.8	8.6	11.4	14.0					99	
using	p-field	9.6	25.2	36.5	45.0	51.7				99		
preceding	s-field	17.9	33.0	44.3	52.6	58.9			99			
coordinate	w-field	1.9	4.4	7.1	9.7	12.4						99

When all the coordinates are considered (upper half of Table 4.1), the percentages are higher than the corresponding percentages for the case where identical fields in adjacent coordinates are omitted (lower half of Table 4.1). This means that the idea of copying certain fields from the preceding coordinate yields to savings which are, for the small values, larger than could have been expected from knowing their distribution in the non-compressed concordance.

Using the information collected from the concordance, all the possible variants for each of the methods in A and B have been checked. Table 4.2 lists for each of the methods the variant for which maximal compression was achieved. The numbers in boldface are the frequent values which are used in methods B and C, the other numbers refer to the lengths of the fields. The value 0 indicates that the field of the preceding coordinate should be copied.

Table 4.2: *Optimal variants of the methods*

Method	p-field	s-field	w-field
A(i)a	0 2 5 10	0 2 5 9	0 4 6 10
A(i)b	1 3 5 10	1 3 5 9	3 5 6 10
A(ii)a	0 1 2 3 4 5 6 10	0 1 2 3 4 5 6 9	0 1 3 4 5 6 7 10
A(ii)b	1 2 3 4 5 6 7 10	1 2 3 4 5 6 7 9	1 2 3 4 5 6 7 10
B(i)	0 **2** 4 10	0 **1** 4 9	0 **4** 6 10
B(ii)	0 **1 2 3** 3 4 5 10	0 **1 2 3** 3 4 5 9	0 **3 4 5** 3 5 6 10
C	0 2 5 10	0 **1 2 3** 3 4 5 9	3 5 6 10

The optimal variants for the methods A(ii) are not surprising: since most of the stored values are small, one could expect the optimal partition to give priority to small field-lengths. For method C, each field is compressed by the best of the other methods, which are A(i)a for the p-field, B(ii) for the s-field and A(i)b for the w-field, thus requiring a header of $1+2+3+2=8$ bits (including one bit for the d-field).

The entries of Table 4.2 were computed using the first refinement mentioned in the description of method D, namely storing $n-1$ instead of n. The second refinement (dropping the leftmost 1) could not be applied, because it is not true that the leftmost bit in every field is a 1. Thus for all the calculations with methods A and B, an integer n was supposed to require $\lfloor \log_2(n-1) \rfloor + 1$ bits for $n > 1$ and one bit for $n = 1$.

As an example for the encoding and decoding processes, consider method C, and a coordinate structure with $(h_d, h_p, h_s, h_w) = (8, 8, 8, 8)$, i.e., one byte for each field. The coordinate we wish to process is $(159, 2, 2, 35)$. Suppose further that only the value in the d-field is the same as in the previous coordinate. Then the length D of the d-field is 0; in the p-field the value 1 is stored, using two bits; nothing is stored in the s-field, because 2 is one of the frequent values and directly referenced by the header; in the w-field the value 34 is stored, using 6 bits. The possible options for the header are numbered from left to right as they appear in Table 4.2, hence the header of this coordinate is 0-10-011-11, where dashes separating the parts corresponding to different fields have been added for clarity; the remaining part of the coordinate is 01-100010. The table $\mathcal{T}$ has $2^8 = 256$ entries; at entry 79 (= 01001111 in binary) the values stored are $(\text{TOT}, P1, S1, W1) = (8, 2, -2, 6)$. When decoding the compressed coordinate 0100111101100010, the leftmost 8 bits are considered as header and converted to the integer 79. Table $\mathcal{T}$ is then accessed with that index, retrieving the 4-tuple $(8, 2, -2, 6)$ which yields the values $(P, S, W) = (2, 0, 6)$. The next TOT = 8 bits are therefore loaded into COOR of size 4 bytes, and after the three shifts we get

$$\text{COOR} = 00000000 - 00000010 - 00000000 - 00100010.$$

Since $\text{TOT} = P + S + W$ the value of the d-field is copied from the last coordinate. Since $P1 < 0$, the value $-S1 = 2$ is put into the s-field.

On our data, the best method was D(i) with an average coordinate length of 3.082 bytes, corresponding to 49% compression relative to the full 6-byte coordinate, and giving a 27% improvement over POM. The next best method was C with 3.14 bytes. Nevertheless, the results depend heavily on the statistics of the specific system at hand, so that for another database, other methods could be preferable.

The main target of the efforts was to try to eliminate or at least reduce the unused space in the coordinates. Note that this can easily be achieved by considering the entire database as a single long run of words, which we could index sequentially from 1 to N, N being the total number of words in the text. Thus $\lfloor \log_2 N \rfloor + 1$ bits would be necessary per coordinate. However, the hierarchical structure is lost, so that, for example, queries asking for the co-occurrence of several words in the same sentence or paragraph are much harder to process. Moreover, when a coordinate is represented by a single, usually large, number, we lose also the possibility to omit certain fields which could be copied from preceding coordinates. A hierarchical structure of a coordinate is therefore preferable for the retrieval algorithms. Some of the new compression methods even outperform the simple method of sequentially numbering the words, since the latter would imply at the RRP database a coordinate length of 26 bits = 3.25 bytes.

4.2 Model Based Concordance Compression

For our model of a textual database, we assume that the text is divided into documents and the documents are made up of words. We thus use only a two level hierarchy to identify the location of a word, which makes the exposition here easier. The methods can, however, be readily adapted to more complex concordance structures, like the 4-level hierarchy mentioned above. In our present model, the conceptual concordance consists, for each word, of a series of (d, w) pairs, d standing for a document number, and w for the index, or offset, of a word within the given document:

$$\begin{array}{ll}
\text{word}_1: & (d_1, w_1)\ (d_1, w_2)\ \cdots (d_1, w_{m_1}) \\
 & (d_2, w_1)\ (d_2, w_2)\ \cdots (d_2, w_{m_2}) \\
 & \cdots \\
 & (d_N, w_1) \cdots (d_N, w_{m_N}) \\
\text{word}_2: & \cdots
\end{array}$$

For a discussion of the problems of relating this conceptual location to a physical location on the disc, see [7].

It is sometimes convenient to translate our 4-level hierarchy to an equivalent one, in which we indicate the index of the next document containing the word, the number of times the word occurs in the document, followed by the list of word indices of the various occurrences:

$$\begin{array}{ll}
\text{word}_1: & (d_1,\ m_1\ ;\ \ w_1, w_2, \ldots, w_{m_1}) \\
 & (d_2,\ m_2\ ;\ \ w_1, \ldots, w_{m_2}) \\
 & \cdots \\
 & (d_N,\ m_N\ ;\ \ w_1, \ldots, w_{m_N}) \\
\text{word}_2: & \cdots
\end{array}$$

Our task is to model each of the components of the latter representation, and use standard compression methods to compress each entity. Below we assume that we know (from the dictionary) the total number of times a word occurs in the database, the number of different documents in which it occurs, and (from a separate table) the number of words in each document. The compression algorithm is then based on predicting the probability distribution of the various values in the coordinates, devising a code based on the predicted distributions, and using the codeword corresponding to the actual value given.

We thus need to generate a large number of codes. If so, the Shannon-Fano method (as defined in [41]) seems the most appropriate if we are concerned with processing speed. Thus an element, which according to the model at hand appears with probability p, will be encoded by $\lceil -\log_2 p \rceil$ bits. Once the length of the codeword is determined, the actual codeword is easily generated. But Shannon-Fano codes are not optimal and might in fact be quite wasteful, especially for the very low probabilities.

While Shannon-Fano coding is fast, when high precision is required Huffman codes are a good alternative. Under the constraint that every codeword consists of an integral number of bits, they are optimal; however their computation is much more involved than that of Shannon-Fano codes, because every codeword depends on the whole set of probabilities. Thus more processing time is needed, but compression is improved. On the other hand, Huffman codes are not effective in the presence of very high probabilities. Elements occurring with high probability have low information content, yet their Huffman codeword cannot be shorter than one bit. If this is a prominent feature, arithmetic coding must be considered.

Arithmetic coding more directly uses the probabilities derived from the model, and overcomes the problem of high probability elements by encoding entire messages, not just codewords. Effectively, an element with probability p is encoded by exactly $-\log_2 p$ bits, which is the information theoretic minimum. While in many contexts arithmetic codes might not improve much on Huffman codes, their superiority here might be substantial, because the model may generate many high probabilities. There is of course a time/space tradeoff, as the computation of arithmetic codes is generally more expensive than that of Huffman codes.

Initially we are at the beginning of the document list and are trying to determine the probability that the next (when we start, this is the first) document containing a term is d documents away from our current location. We know the number of documents that contain the term, say N, and the number of documents, say D, from which these are chosen. (More generally, after we have located a number of documents that contain the term, D and N will respectively represent the total number of remaining documents, and, of these, the number that contain the term. Our reasoning will then continue in parallel to that of the first occurrence.)

Our first question, then, is: what is the probability distribution of the first/next document containing the term. Assuming that the events involved are independent and equally distributed, this is equivalent to asking, if N different objects are selected at random from an ordered set of D objects, what is the probability that d is the index of the object with minimum index?

Because of the uniformity assumption, each of the $\binom{D}{N}$ ways of picking N out of D objects have same probability, viz, $1/\binom{D}{N}$. But of these, only $\binom{D-d}{N-1}$ satisfy the condition that d is the minimum index. That is, certainly one document must be the d-th one, so we only have freedom to choose $N-1$ additional documents. Since all of these must have index greater than d, we have only $D-d$ options for these $N-1$ selections. Thus the probability that the next document has (relative) position d is $\Pr(d) = \binom{D-d}{N-1}/\binom{D}{N}$.

We first note this is a true probability:

$$\sum_d \Pr(d) = \sum_{d=1}^{D-N+1} \frac{\binom{D-d}{N-1}}{\binom{D}{N}} = \sum_{k=N-1}^{D-1} \frac{\binom{k}{N-1}}{\binom{D}{N}} = \frac{\binom{D}{N}}{\binom{D}{N}} = 1,$$

where the last equality uses the well known combinatoric identity that permits summation over the upper value in the binomial coefficient [11]. Second, we note that we can rewrite the probability as

$$\left(\frac{N}{D-N+1}\right) \times \left(1-\frac{d}{D}\right) \times \left(1-\frac{d}{D-1}\right) \times \cdots \times \left(1-\frac{d}{D-N+2}\right).$$

If $d \ll D$, this is approximately $(N/D) \times (1-d/D)^{N-1}$, which is in turn approximately proportional to $e^{-d(N-1)/D}$ or γ^d, for $\gamma = e^{-(N-1)/D}$. This last form is that of the geometric distribution recommended by Witten et al.[42].

The encoding process is then as follows. We wish to encode the d-field of the next coordinates $(d, m;\ w_1, \ldots, w_m)$. Assuming that the probability distribution of d is given by $\Pr(d)$, we construct a code based on $\{\Pr(d)\}_{d=1}^{D-N+1}$. This assigns codewords to all the possible values

```
{
    for $s \longleftarrow 1$ to $S$      /* for each word in concordance */
        $D$   $\longleftarrow$   total number of documents
        $T$   $\longleftarrow$   total number of occurrences of word $s$
        $N$   $\longleftarrow$   total number of documents in which word $s$ occurs
        $d_0$  $\longleftarrow$   $0$
        for $i \longleftarrow 1$ to $N$      /* for each document containing word $s$ */
        {
            /* process document $i$ */
            output d_code($d_i - d_{i-1}, N - i, D$)
            if $T > N$
                output m_code($m_i - 1$,  $(T - N)/N$,  $T - N$)
            /* process occurrences of word $s$ in document $i$ */
            $W$   $\longleftarrow$   total number of words in document $i$
            $w_0$  $\longleftarrow$   $0$
            for $j \longleftarrow 1$ to $m_i$
            {
                output w_code($w_j - w_{j-1}, m_i - j, W$)
                $W$   $\longleftarrow$   $W - (w_j - w_{j-1})$
            }
            /* update parameters and continue */
            $D$   $\longleftarrow$   $D - (d_i - d_{i-1})$
            $T$   $\longleftarrow$   $T - m_i$
        }
    }
}
d_code($d, N, D$)
{
    construct a code $\mathcal{C}_1$ based on probabilities that $d = k$: $\left\{\binom{D-k}{N}/\binom{D}{N+1}\right\}_{k=1}^{D-i+1}$
    return $\mathcal{C}_1(d)$
}
m_code($x, \lambda, max$)
{
    $F$   $\longleftarrow$   $\sum_{k=0}^{max} e^{-\lambda}\frac{\lambda^k}{k!}$      /* correction factor for truncated Poisson distribution */
    construct a code $\mathcal{C}_2$ based on probabilities that $x = k$: $\left\{\frac{1}{F}e^{-\lambda}\frac{\lambda^k}{k!}\right\}_{k=0}^{max}$
    return $\mathcal{C}_2(x)$
}
w_code($w, m, W$)
{
    construct a code $\mathcal{C}_3$ based on probabilities that $w = k$: $\left\{\binom{W-k}{m}/\binom{W}{m+1}\right\}_{k=1}^{W-i+1}$
    return $\mathcal{C}_3(w)$
}
```

FIGURE 4.3: *Concordance compression algorithm*

of d, from which we use the codeword corresponding to the actual value d in our coordinate. If the estimate is good, the actual value d will be assigned a high probability by the model, and therefore be encoded with a small number of bits.

Next we encode the number of occurrences of the term in this document. Let us suppose that we have T occurrences of the term remaining (initially, this will be the total number of occurrences of the term in the database). The T occurrences are to be distributed into the N remaining documents the word occurs in. Thus we know that each document being considered must have at least a single term, that is, $m = 1+x$, where $x \geq 0$. If $T = N$, then clearly $x = 0$ ($m = 1$), and we need output no code — m conveys no information in this case. If $T > N$, then we must distribute the $T - N$ terms not accounted for over the remaining N documents that contain the term. We assume, for simplicity, that the additional amount, x, going to the currently considered document is Poisson distributed, with mean $\lambda = (T - N)/N$. The Poisson distribution is given by $\Pr(x) = e^{-\lambda}\frac{\lambda^x}{x!}$. This allows us to compute the probability of x for all possible values ($x = 0, 1, \ldots, T - N$) and to then encode x using one of the encodings above.

We must finally encode all the m offsets. But this problem is formally identical to that of encoding the next document. The current document has W words, so the distribution of w, the first occurrence of the word, is given by the probabilities $\binom{W-w}{m-1}/\binom{W}{m}$. Once this is encoded, we have a problem identical to the initial one in form, except that we now have $m - 1$ positions left to encode and $W - w$ locations. This continues until the last term, which is uniformly distributed over the remaining word locations.

Then we encode the next document, but this is again a problem identical in form to the initial problem—only we now have one fewer document ($N - 1$) having the term, and d fewer target documents ($D - d$) to consider.

The formal encoding algorithm is given in Figure 4.3. We begin with a conceptual concordance, represented for the purpose of this algorithm as a list of entries. Our concordance controls S different words. For each word, there is an entry for each document it occurs in, of the form $(d_i, m_i;\ w_1, \ldots, w_{m_i})$, where d_i, m_i and w_j are given similarly to the representation (2) defined above.

Note that we do not encode the absolute values d_i and w_j, but the relative increases $d_i - d_{i-1}$ and $w_j - w_{j-1}$; this is necessary, because we redefine, in each iteration, the sizes D, W and T to be the *remaining* number of documents, number of words in the current document, and number of occurrences of the current word, respectively.

In fact, one should also deal with the possibility where the independence assumptions of the previous section are not necessarily true. In particular, we consider the case where terms cluster not only within a document, but even at the between document level. Details of this model can be found in [43].

5. BITMAPS

For every distinct word W of the database, a bit-map $B(W)$ is constructed, which acts as "occurrence"-map at the document level. The length (in bits) of each map is the number of documents in the system. Thus, in the RRP for example, the length of each map is about 6K bytes. These maps are stored in compressed form on a secondary storage device. At RRP, the compression algorithm was taken from [44], reducing the size of a map to 350 bytes on the average. This compression method was used for only about 10% of the words, those which appear at least 70 times; for the remaining words, the *list* of document numbers is kept and transformed into bit-map form at processing time. The space needed for the bit-map file in its entirety is 33.5 MB, expanding the overall space requirement of the entire retrieval system by about 5%.

At the beginning of the process dealing with a query of the type given in eqn. (1.1), the maps $B(A_{ij})$ are retrieved, for $i = 1, \ldots, m$ and $j = 1, \ldots, n_i$. They are decompressed and a new map ANDVEC is constructed:

$$\text{ANDVEC} = \bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{n_i} B(A_{ij}) \right).$$

The bit-map ANDVEC serves as a "filter", for only documents corresponding to 1-bits in ANDVEC can possibly contain a solution. Note that no more than three full-length maps are simultaneously needed for its construction.

For certain queries, in particular when keywords with a small number of occurrences in the text are used, ANDVEC will consist only of zeros, which indicates that nothing should be retrieved. In such cases the user gets the correct if somewhat meager results, without a single merge or collate action having been executed. But even if ANDVEC is not null, it will usually be much sparser than its components. These maps can improve the performance of the retrieval process in many ways to be now described.

5.1 Usefulness of Bitmaps in IR

First, bit-maps can be helpful in reducing the number of I/O operations involved in the query-processing phase. Indeed, since the concordance file is usually too big to be stored in the internal memory, it is kept in compressed form on secondary storage, and parts of it are fetched when needed and decompressed. The compressed concordance file is partitioned into equi-sized blocks such that one block can be read by a single I/O operation; it is accessed via the dictionary, which contains for each word a pointer to the corresponding (first) block. A block can contain coordinates of many "small" words (i.e., words with low frequency in the database), but on the other hand, the coordinate list of a single "large" (high-frequency) word may extend over several consecutive blocks. In the RRP, for example, about half of the words appear only once, but on the other hand there are some words that occur hundreds of thousands of times! It is for the large words that the bit-map ANDVEC may lead to significant savings in the number of I/O operations. Rather than reading *all* the blocks to collect the list of coordinates which will later be merged and/or collated, we access only blocks which contain coordinates in the documents specified by the 1-bits of ANDVEC. Hence if the map is sparse enough, only a small *subset* of the blocks need to be fetched and decompressed. To implement this idea, we need, in addition to the bit-map, also a small list $L(W)$ for each large word W, $L(W) = \{(f_j, \ell_j)\}$, where f_j and ℓ_j are respectively the document numbers of the first and last coordinate of W in block number j, and j runs over the indices of blocks which contain coordinates of W. The list $L(W)$ is scanned together with the bit-map, and if there is no 1-bit in ANDVEC in the bit-range $[f_j, \ell_j]$, the block j is simply skipped.

There are, however, savings beyond I/O-operations. Once a concordance block containing some coordinates which might be relevant is read, it is scanned in parallel with ANDVEC. Coordinates with document numbers corresponding to 0-bits are skipped. For the axis, which is the first keyword A_i to be handled, this means that only parts of the lists $\mathcal{C}(A_{ij})$ will be transferred to a working area, where they are merged. In order to save internal memory space during the query processing, the lists of the keywords A_{kj}, for $k \neq i$, are not merged like the lists of the axis, but are directly collated with the axis. Such collations can be involved operations, as the distance constraints may cause each coordinate of the axis to be checked against several coordinates of every variant of other keywords, and conversely every such coordinate might collate with several coordinates of the axis. Therefore the use of ANDVEC may save time by reducing the number of collations. Moreover, after all the variants of the second keyword have been collated with the axis, the coordinates of the axis which were not matched can be rejected, so that the axis may shrink considerably. Now ANDVEC can be updated by deleting some of its 1-bits, which again tends to reduce the number of read operations and collations when handling the following keywords. The updates of the axis and ANDVEC are repeated after the processing of each keyword A_j of the query (1.1).

For conventional query processing algorithms, the consequence of increasing the number m of keywords is an increased processing time, whereas the set of solutions can only shrink. When m is increased with the bit-map approach, however, the time needed to retrieve the maps and to perform some additional logical operations is usually largely compensated for by the savings in I/O operations caused by a sparser ANDVEC. The new approach seems thus to be particularly attractive for a large number of keywords. Users are therefore encouraged to change their policy and to submit more complex queries!

Another possible application of the bit-maps is for getting a selective display of the results. A user is often not interested in finding *all* the occurrences of a certain phrase in the database, as specified by the query, but only in a small subset corresponding to a certain author or a certain period. The usual way to process such special requests consists in executing first the search ignoring the restrictions, and then filtering out the solutions which are not needed. This can be very wasteful and time-consuming, particularly if the required sub-range (period or author(s)) is small. The bit-maps allow the problem to be dealt with in a natural way, requiring only minor changes to adapt the search program to this application. All we need is to prepare a small repertoire $\mathcal{R}$ of fixed bit-maps, say one for each author, where the 1-bits indicate the documents written by this author, and a map for the documents of each year or period, etc. The restrictions can now be formulated at the same time the query is submitted. In the first line of the above algorithm, ANDVEC will not be initialized by a string containing only 1's, but by a logical combination of elements of $\mathcal{R}$, as induced by the additional restrictions. Thus user-imposed restrictions on required ranges to which solutions should belong on one hand, and query-imposed restrictions on the co-occurrence of keywords on the other, are processed in exactly the same way, resulting in a bit-vector, the sparsity of which depends directly on the severity of the restrictions. As was pointed out earlier, this may lead to savings in processing time and I/O operations.

Finally, bit-maps can be also helpful in handling negative keywords. If a query including some negative keywords D_i is submitted at the document-level, one can use the binary complements $\overline{B(D_i)}$ of the maps, since only documents with no occurrence of D_i (indicated by the 0-bits) can be relevant. However, for other levels, the processing is not so simple. In fact, if the query is not on the document level, the bit-maps of the negative keywords are useless, and ANDVEC is formed only by the maps of the positive keywords. This difference in the treatment of negative and positive keywords is due to the fact that a 0-bit in the bit-vector of a positive keyword means that the corresponding document cannot possibly be relevant, whereas a 1-bit in the bit-vector of a negative keyword D_j only implies that D_j appears in the corresponding document; however, this document can still be retrieved, if D_j is not in the specified neighborhood of the other keywords. Nevertheless, even though the negative keywords do not contribute in rendering ANDVEC sparser, ANDVEC will still be useful also for the negative words: only coordinates in the relevant documents have to be checked *not* to fall in the vicinity of the axis, as imposed by the (l_i, u_i).

5.2 Compression of Bitmaps

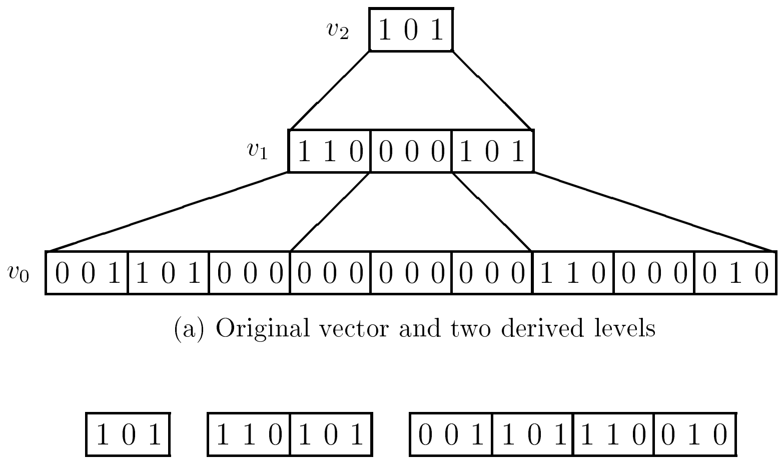
It would be wasteful to store the bit-maps in their original form, since they are usually very sparse (the great majority of the words occur in very few documents). Schuegraf [45] proposes to use *run-length coding* for the compression of sparse bit-vectors, in which a string of consecutive zeros terminated by a one (called a *run*) is replaced by the length of the run. A sophisticated run-length coding technique can be found in Teuhola [46]. Jakobsson [47] suggests to partition each vector into k-bit blocks, and to apply Huffman coding on the 2^k possible bit-patterns. This method is referred below as method `NORUN`.

5.2.1 Hierarchical compression

In this section we concentrate on *hierarchical bit-vector compression*: let us partition the original bit-vector v_0 of length l_0 bits into k_0 equal blocks of r_0 bits, $r_0 \cdot k_0 = l_0$, and drop the blocks consisting only of zeros. The resulting sequence of non-zero blocks does not allow the reconstruction of v_0, unless we add a list of the indices of these blocks in the original vector. This list of up to k_0 indices is kept as a binary vector v_1 of $l_1 = k_0$ bits, where there is a 1 in position i if and only if the i-th block of v_0 is not all zero. Now v_1 can further be compressed by the same method.

In other words, a sequence of bit-vectors v_j is constructed, each bit in v_j being the result of ORing the bits in the corresponding block in v_{j-1}. The procedure is repeated recursively until a level t is reached where the vector length reduces to a few bytes, which will form a single block. The compressed form of v_0 is then obtained by concatenating all the nonzero blocks of the various v_i, while retaining the block-level information. Decompression is obtained simply by reversing these operations and their order. We start at level t, and pass from one level to the next by inserting blocks of zeros into level $j-1$ for every 0-bit in level j.

Figure 5.1 depicts an example of a small vector v_0 of 27 bits and its derived levels v_1 and v_2, with $r_i = 3$ for $i = 0, 1, 2$ and $t = 2$. The sizes r_j of the blocks are parameters and



(a) Original vector and two derived levels

(b) Compressed vector

FIGURE 5.1: *Hierarchical bit-vector compression*

can change from level to level for a given vector, and even from one word of the database to another, although the latter is not practical for our applications. Because of the structure of the compressed vector, we call this the `TREE` method, and shall use in our discussion the usual tree-vocabulary: the *root* of the tree is the single block on the top level, and for a block x in v_{j+1} which is obtained by ORing the blocks $y_1, \ldots, y_{r_j}$ of v_j, we say that x is the *parent* of the non-zero blocks among the y_i.

The `TREE` method was proposed by Wedekind & Härder [48]. It appears also in Vallarino [49], who used it for two-dimensional bit-maps, but only with one level of compression. In [50], the parameters (block size and height of the tree) are chosen assuming that the bit-vectors are generated by a memoryless information source, i.e., each bit in v_0 has a constant probability p_0 for being 1, independently from each other. However, for bit-maps in information retrieval systems, this assumption is not very realistic a priori, as adjacent bits often represent documents written by the same author; there is a positive correlation for a word to appear in consecutive documents, because of the specific style of the author or simply because such documents often treat the same or related subjects.

We first remark that the hierarchical method does not always yield real compression. Consider for example a vector v_0 for which the indices of the 1-bits are of the form ir_0 for $i \leq l_0/r_0$. Then there are no zero-blocks (of size r_0) in v_0, moreover all the bits of v_i for $i > 0$ will be 1, so that the whole tree must be kept. Therefore the method should be used only for sparse vectors.

In the other extreme case, when v_0 is very sparse, the `TREE` method may again be wasteful: let $d = \lceil \log_2 l_0 \rceil$, so that a d-bit number suffices to identify any bit-position in v_0. If the vector is extremely sparse, we could simply list the positions of all the 1-bits, using d bits for each. This is in fact the inverse of the transformation performed by the bit-vectors: basically, for every different word W of the database, there is one entry in the inverted file containing the list of references of W, and this list is transformed into a bit-map; here we change the bit-map back into its original form of a list.

A small example will illustrate how the bijection of the previous paragraph between lists and bit-maps can be used to improve method TREE. Suppose that among the $r_0 \cdot r_1 \cdot r_2$ first bits of v_0 only position j contains a one. The first bit in level 3, which corresponds to the ORing of these bits, will thus be set to 1 and will point to a sub-tree consisting of three blocks, one on each of the lower levels. Hence in this case a single 1-bit caused the addition of at least $r_0 + r_1 + r_2$ bits to the compressed map, since if it were zero, the whole sub-tree would have been omitted. We conclude that if $r_0 + r_1 + r_2 \geq d$, it is preferable to consider position j as containing zero, thus omitting the bits of the sub-tree, and to add the number j to an appended list L, using only d bits. This example is readily generalized so as to obtain an optimal partition between tree and list for every given vector, as will now be shown.

We define l_j and k_j respectively as the number of bits and the number of blocks in v_j, for $0 \leq j \leq t$. Note that $r_j \cdot k_j = l_j$. Denote by $T(i,j)$ the sub-tree rooted at the i-th block of v_j, with $0 \leq j \leq t$ and $1 \leq i \leq k_j$. Let $S(i,j)$ be the size in bits of the compressed form of the sub-tree $T(i,j)$, i.e., the total number of bits in all the non-zero blocks in $T(i,j)$, and let $N(i,j)$ be the number of 1-bits in the part of the original vector v_0 which belongs to $T(i,j)$.

During the bottom-up construction of the tree these quantities are recursively evaluated for $0 \leq j \leq t$ and $1 \leq i \leq k_j$ by:

$$N(i,j) = \begin{cases} \text{number of 1-bits in block } i \text{ of } v_0 & \text{if } j = 0, \\ \sum_{h=1}^{r_j} N\left((i-1)r_j + h, j-1\right) & \text{if } j > 0; \end{cases}$$

$$S(i,j) = \begin{cases} 0 & \text{if } j = 0 \text{ and } T(i,0) \text{ contains only zeros}, \\ r_0 & \text{if } j = 0 \text{ and } T(i,0) \text{ contains a 1-bit}, \\ r_j + \sum_{h=1}^{r_j} S\left((i-1)r_j + h, j-1\right) & \text{if } j > 0. \end{cases}$$

At each step, we check the condition

$$d \cdot N(i,j) \leq S(i,j). \tag{5.1}$$

If it holds, we prune the tree at the root of $T(i,j)$, adding the indices of the $N(i,j)$ 1-bits to the list L, and setting then $N(i,j)$ and $S(i,j)$ to zero. Hence the algorithm partitions the set of 1-bits into two disjoint subsets: those which are compressed by the TREE-method and those kept as a list. In particular, if the pruning action takes place at the only block of the top level, there will be no tree at all.

Note that by definition of $S(i,j)$, the line corresponding to the case $j > 0$ should in fact be slightly different: r_j should be added to the sum $X = \sum_{h=1}^{r_j} S((i-1)r_j + h, j-1)$ only if $X \neq 0$. However, no error will result from letting the definition in its present form. Indeed, if $X = 0$, then also $N(i,j) = 0$ so that the inequality in (5.1) is satisfied in this case, thus $S(i,j)$ will anyway be set to zero. Note also that in case of equality in (5.1), we execute a pruning action although a priori there is no gain. However, since the number of 1-bits in v_j is thereby reduced, this may enable further prunings in higher levels, which otherwise might not have been done.

We now further compress the list L (of indices of 1-bits which were "pruned" from the tree) using POM, which can be adapted to the compression of a list of d-bit numbers: we choose an integer $c < d - 1$ as parameter, and form a bit-map v of $k = \lceil l_0/2^c \rceil$ bits, where bit i, for $0 \leq i < k$, is set to 1 if and only if the integer i occurs in the $d - c$ leftmost bits of at least one number in L. Thus a 1-bit in position i of v indicates that there are one or more numbers in L in the range $[i2^c, (i+1)2^c - 1]$. For each 1-bit in v, the numbers of the corresponding range can now be stored as relative indices in that range, using only c bits for each, and an additional bit per index serving as flag, which identifies the last index of each range. Further compression of the list L is thus worthwhile only if

$$d \cdot |L| \; > \; k + (c+1)|L|. \tag{5.2}$$

The left hand side of (5.2) corresponds to the number of bits needed to keep the list L uncompressed. Therefore this secondary compression is justified only when the number of elements in L exceeds $k/(d-c-1)$.

For example, for $l_0 = 128$ and $c = 5$, there are 4 blocks of 2^5 bits each; suppose the numbers in L are 36, 50, 62, 105 and 116 (at least five elements are necessary to justify further compression). Then there are three elements in the second block, with relative indices 4, 18 and 30, and there are two elements in the fourth block, with relative indices 9 and 20, the two other blocks being empty. This is shown in Figure 5.2.

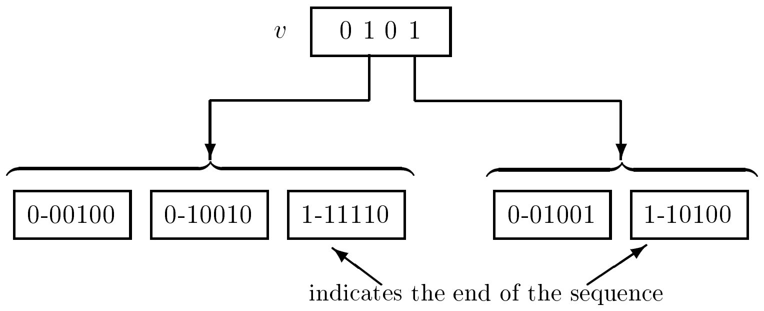


FIGURE 5.2: *Further compression of index list*

Finally we get even better compression by adapting the cut-off condition (5.1) dynamically to the number of elements in L. During the construction of the tree, we keep track of this number and as soon as it exceeds $k/(d-c-1)$, i.e., it is worthwhile to further compress the list, we can relax the condition in (5.1) to

$$(c+1) \cdot N(i,j) \leq S(i,j), \tag{5.3}$$

since any index which will be added to L, will use only $c+1$ bits for its encoding.

In fact, after recognizing that L will be compressed, we should check again the blocks already handled, since a sub-tree $T(i,j)$ may satisfy (5.3) without satisfying (5.1). Nevertheless, we have preferred to keep the simplicity of the algorithm and not to check again previously handled blocks, even at the price of losing some of the compression efficiency. Often, there will be no such loss, since if we are at the top level when $|L|$ becomes large enough to satisfy (5.2), this means that the vector v_0 will be kept in its entirety as a list. If we are not at the top level, say at the root of $T(i,j)$ for $j < t$, then all the previously handled trees will be reconsidered as part of larger trees, which are rooted on the next higher level. Hence it is possible that the sub-tree $T(i,j)$, which satisfies (5.3) but not (5.1) (and thus was not pruned at level j), will be removed as part of a larger sub-tree rooted at level $j+1$.

5.2.2 Combining Huffman and run-length coding

As we are interested in sparse bit-strings, we can assume that the probability p of a block of k consecutive bits being zero is high. If $p \geq 0.5$, method NORUN assigns to this 0-block a codeword of length one bit, so we can never expect a better compression factor than k. On the other hand, k cannot be too large since we must generate codewords for 2^k different blocks.

In order to get a better compression, we extend the idea of method NORUN in the following way: there will be codewords for the $2^k - 1$ non-zero blocks of length k, plus some additional codewords representing runs of zero-blocks of different lengths. In the sequel, we use the term 'run' to designate a run of zero-blocks of k bits each.

The length (number of k-bit blocks) of a run can take any value up to l_0/k, so it is impractical to generate a codeword for each: as was just pointed out, k cannot be very large, but l_0 is large for applications of practical importance. On the other hand, using a fixed-length code for the run length would be wasteful since this code must suffice for the maximal length, while most of the runs are short. The following methods attempt to overcome these difficulties.

Starting with a fixed-length code for the run-lengths, we like to get rid of the leading zeros in the binary representation $B(\ell)$ of run-length ℓ, but we clearly cannot simply omit them, since this would lead to ambiguities. We *can* omit the leading zeros if we have additional information such as the position of the leftmost 1 in $B(\ell)$. Hence, partition the possible lengths into classes C_i, containing run-lengths ℓ which satisfy $2^{i-1} \leq \ell < 2^i$, $i = 1, \ldots, \lfloor \log_2(l_0/k) \rfloor$. The $2^k - 1$ non-zero block-patterns and the classes C_i are assigned Huffman codewords corresponding to the frequency of their occurrence in the file; a run of length ℓ belonging to class C_i is encoded by the codeword for C_i, followed by $i - 1$ bits representing the number $\ell - 2^{i-1}$. For example, a run of 77 0-blocks is assigned the codeword for C_7 followed by the 6 bits 001101. Note that a run consisting of a single 0-block is encoded by the codeword for C_1, without being followed by any supplementary bits.

The Huffman decoding procedure has to be modified in the following way: The table contains for every codeword the corresponding class C_i as well as $i - 1$. Then, when the codeword which corresponds to class C_i is identified, the next $i - 1$ bits are considered as the binary representation of an integer m. The codeword for C_i followed by those $i - 1$ bits represent together a run of length $m + 2^{i-1}$; the decoding according to Huffman's procedure resumes at the i-th bit following the codeword for C_i. Summarizing, we in fact encode the length of the binary representation of the length of a run, and the method is henceforth called LLRUN.

Method LLRUN seems to be efficient since the number of bits in the binary representation of integers is reduced to a minimum and the lengths of the codewords are optimized by Huffman's algorithm. But encoding and decoding are admittedly complicated and thus time consuming. We therefore propose other methods for which the encoded file will consist only of codewords, each representing a certain string of bits. Even if their compression factor is lower than LLRUN's, these methods are justified by their simpler processing.

To the $2^k - 1$ codewords for non-zero blocks, a set S of t codewords is adjoined representing $h_0, h_1, \ldots, h_{t-1}$ consecutive 0-blocks. Any run of zero-blocks will now be encoded by a suitable linear combination of some of these codes. The number t depends on the numeration system according to which we choose the h_i's and on the maximal run-length M, but should be low compared to 2^k. Thus in comparison with method `NORUN`, the table used for compressing and decoding should only slightly increase in size, but long runs are handled more efficiently. The encoding algorithm now becomes:

Step 1: Collect statistics on the distribution of run-lengths and on the set NZ of the $2^k - 1$ possible non-zero blocks. The total number of occurrences of these blocks is denoted by N_0 and is fixed for a given set of bit-maps.

Step 2: Decompose the integers representing the run-lengths in the numeration system with set S of "basis" elements; denote by $\mathrm{TNO}(S)$ the total number of occurrences of the elements of S.

Step 3: Evaluate the relative frequency of appearance of the $2^k - 1 + t$ elements of $\mathrm{NZ} \cup S$ and assign a Huffman code accordingly.

For any $x \in (\mathrm{NZ} \cup S)$, let $p(x)$ be the probability of the occurrence of x and $\ell(x)$ the length (in bits) of the codeword assigned to x by the Huffman algorithm. The weighted average length of a codeword is then given by $\mathrm{AL}(S) = \sum_{x\in(\mathrm{NZ}\cup S)} p(x)\ell(x)$ and the size of the compressed file is

$$\mathrm{AL}(S) \times (N_0 + \mathrm{TNO}(S)).$$

After fixing k so as to allow easy processing of k-bit blocks, the only parameter in the algorithm is the set S. In what follows, we propose several possible choices for the set $S = \{1 = h_0 < h_1 < \ldots < h_{t-1}\}$. To overcome coding problems, the h_i and the bounds on the associated digits a_i should be so that there is a unique representation of the form $L = \sum_i a_i h_i$ for every natural number L.

Given such a set S, the representation of an integer L is obtained by the following simple procedure:

```
**for** $i \leftarrow t-1$ **to** $0$ **by** $-1$
    $a_i \leftarrow \lfloor L/h_i \rfloor$
    $L \leftarrow L - a_i \times h_i$
**end**
```

The digit a_i is the number of times the codeword for h_i is repeated. This algorithm produces a representation $L = \sum_{i=0}^{t-1} a_i h_i$ which satisfies

$$\sum_{i=0}^{j} a_i h_i < h_{j+1} \qquad \text{for} \quad j = 0, \ldots, t-1. \tag{5.4}$$

Condition (5.4) guarantees uniqueness of representation (see [51]).

A natural choice for S is the standard binary system (method `POW2`), $h_i = 2^i$, $i \geq 0$, or higher base numeration systems such as $h_i = m^i$, $i \geq 0$ for some $m > 2$. If the run-length is

L it will be expressed as $L = \sum_i a_i m^i$, with $0 \le a_i < m$ and if $a_i > 0$, the codeword for m^i will be repeated a_i times. Higher base systems can be motivated by the following reason.

If p is the probability that a k-bit block consists only of zeros, then the probability of a run of r blocks is roughly $p^r(1-p)$, i.e., the run-lengths have approximately geometric distribution. The distribution is not exactly geometric since the involved events (some adjacent blocks contain only zeros, i.e., a certain word does not appear in some consecutive documents) are not independent. Nevertheless the experiments showed that the number of runs of a given length is an exponentially decreasing function of run-length (see Figure 2.1 below). Hence with increasing base of the numeration systems, the relative weight of the h_i for small i will rise, which yields a less uniform distribution for the elements of $\mathrm{NZ} \cup S$ calculated in Step 3. This has a tendency to improve the compression obtained by the Huffman codes. Therefore passing to higher order numeration systems will reduce the value of $\mathrm{AL}(S)$.

On the other hand, when numeration systems to base m are used, $\mathrm{TNO}(S)$ is an increasing function of m. Define r by $m^r \le M < m^{r+1}$ so that at most r m-ary digits are required to express a run-length. If the lengths are uniformly distributed, the average number of basis elements needed (counting multiplicities) is proportional to $(m-1)r = (m-1)\log_m M$, which is increasing for $m > 1$, and this was also the case for our nearly geometric distribution. Thus from this point of view, lower base numeration systems are preferable.

As an attempt to reduce $\mathrm{TNO}(S)$, we pass to numeration systems with special properties, such as systems based on Fibonacci numbers

$$F_0 = 0, \quad F_1 = 1, \qquad F_i = F_{i-1} + F_{i-2} \quad \text{for} \quad i \ge 2.$$

(a) The binary Fibonacci numeration system (method `FIB2`): $h_i = F_{i+2}$. Any integer L can be expressed as $L = \sum_{i \ge 0} b_i F_{i+2}$ with $b_i = 0$ or 1, such that this binary representation of L consisting of the string of b_i's contains no adjacent 1's. This fact for a binary Fibonacci system is equivalent to condition (5.4), and reduces the number of codewords we need to represent a specific run-length, even though the number of added codewords is larger than for `POW2` (instead of $t(\texttt{POW2}) = \lfloor \log_2 M \rfloor$ we have $t(\texttt{FIB2}) = \lfloor \log_\phi(\sqrt{5}M) \rfloor - 1$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio). For example, when all the run-lengths are equally probable, the average number of codewords per run is asymptotically (as $k \to \infty$) $\frac{1}{2}(1 - 1/\sqrt{5})t(\texttt{FIB2})$ instead of $\frac{1}{2}t(\texttt{POW2})$.

(b) A ternary Fibonacci numeration system: $h_i = F_{2(i+1)}$, i.e., we use only Fibonacci numbers with even indices. This system has the property that there is at least one 0 between any two 2's. This fact for a ternary Fibonacci system is again equivalent to (5.4).

6. FINAL REMARKS

Modern Information Retrieval Systems are generally based on inverted files and require large amounts of storage space and powerful machines for the processing of sophisticated queries. Data compression techniques that are specifically adapted to the various files in an IR environment can improve the performance, both by reducing the space needed to store the numerous auxiliary files, and by reducing the necessary data transfer and thereby achieving a speedup.

We have presented a selected choice of techniques pertaining to the different files involved in a full-text IR system; some are given with considerable detail, others are only roughly described. We hope that, nevertheless, the reader will get a useful overall picture, which can be completed by means of the appended literature.

Our main focus has been on IR systems using inverted files. With the development of ever more powerful computers, it may well be that brute force methods, like searching large files using some pattern matching techniques (see, e.g., [52]) or probabilistic approaches using signature files (see, e.g., [53]), will again be considered a feasible alternative, even for very large files.

References

[1] W.B. Pennebaker, J.L. Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, New York (1993).

[2] P. Bratley, Y. Choueka, *Inf. Processing & Management* **18** (1982) 257–266.

[3] R. Attar, Y. Choueka, N. Dershowitz, A.S. Fraenkel, *J. ACM* **25** (1978) 52–66.

[4] A. Bookstein, S.T. Klein, *Information Processing & Management* **26** (1990) 525–533.

[5] D.R. Davis, A.D. Lin, *Comm. ACM* **8** (1965) 243–246.

[6] Y. Choueka, A.S. Fraenkel, S.T. Klein, E. Segal, *Proc. 10-th ACM-SIGIR Conf.,* New Orleans (1987) 306–315.

[7] A. Bookstein, S.T. Klein, D.A. Ziff, *Information Processing & Management* **28** (1992) 795–806.

[8] A.S. Fraenkel, *Jurimetrics J.* **16** (1976) 149–156.

[9] S. Even, *IEEE Trans. on Inf. Theory* **IT–9** (1963) 109–112.

[10] B. McMillan, *IRE Trans. on Inf. Th.* **IT–2** (1956) 115–116.

[11] S. Even, *Graph Algorithms*, Computer Science Press (1979).

[12] C.E. Shannon, *Bell System Tech. J.*, **27** (1948) 379–423, 623–656.

[13] D. Huffman, *Proc. of the IRE* **40** (1952) 1098–1101.

[14] J. Van Leeuwen, *Proc. 3rd ICALP Conference*, Edinburgh University Press (1976) 382–410.

[15] A.S. Fraenkel, S.T. Klein, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[16] D.E. Knuth, *The Art of Computer Programming, Vol* **III**, *Sorting and Searching,* Addison-Wesley, Reading, MA (1973).

[17] D.E. Knuth, *The Art of Computer Programming, Vol* **I**, *Fundamental Algorithms,* Addison-Wesley, Reading, MA (1973).

[18] L.L. Larmore, D.S. Hirschberg, *Journal ACM* **37** (1990) 464–473.

[19] A.S. Fraenkel, S.T. Klein, *The Computer Journal* **36** (1993) 668–678.

[20] H.K. Reghbati, *Computer* **14**, Issue 4 (1981) 71–76.

[21] A. Bookstein, S.T. Klein, *ACM Trans. on Information Systems* **8** (1990) 27–49.

[22] A. Moffat, A. Turpin, J. Katajainen, *Proc. Data Compression Conference DCC–95*, Snowbird, Utah (1995) 192–201.

[23] E.S. Schwartz, B. Kallick, *Comm. of the ACM* **7** (1964) 166–169.

[24] D.S. Hirschberg, D.A. Lelewer, *Comm. of the ACM* **33** (1990) 449–459.

[25] I.H. Witten, A. Moffat, T.C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images,* Van Nostrand Reinhold, New York (1994).

[26] A. Moffat, A. Turpin, *Proc. Data Compression Conference DCC–96*, Snowbird, Utah (1996) 182–191.

[27] E.N. Gilbert, E.F. Moore, *The Bell System Technical Journal* **38** (1959) 933–968.

[28] A. Bookstein, S.T. Klein, *Computing* **50** (1993) 279–296.

[29] G.K. Zipf, *The Psycho-Biology of Language*, Boston, Houghton (1935).

[30] G.H.O. Katona, T.O.H. Nemetz, *IEEE Trans. on Inf. Th.* **IT–11** (1965) 284–292.

[31] A. Moffat, J. Zobel, N. Sharman, *IEEE Transactions on Knowledge and Data Engineering* **9** (1997) 302–313.

[32] J.J. Rissanen, *IBM J. Res. Dev.* **20** (1976) 198–203.

[33] J. Rissanen, G.G. Langdon, *IBM J. Res. Dev.* **23** (1979) 149–162.

[34] I.H Witten, R.M. Neal, J.G. Cleary, *Comm. of the ACM* **30** (1987) 520–540.

[35] A.S. Fraenkel, M. Mor, Y. Perl, *Acta Informatica* **20** (1983) 371–389.

[36] A.S. Fraenkel, M. Mor, Y. Perl, *Proc. 19th Allerton Conf. on Communication, Control and Computing* (1981) 762–768.

[37] J. Ziv, A. Lempel, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.

[38] J. Ziv, A. Lempel, *IEEE Trans. on Inf. Th.* **IT–24** (1978) 530–536.

[39] D.L. Whiting, G.A. George, G.E. Ivey, U.S. Patent 5,126,739 (1992).

[40] A.S. Fraenkel, M. Mor, *The Computer Journal* **26** (1983) 336–343.

[41] R.W. Hamming, *Coding and Information Theory,* Prentice-Hall, Englewood Cliffs, NJ (1980).

[42] I.H. Witten, T.C. Bell, C.G. Nevill, *Proc. Data Compression Conference,* Snowbird, Utah (1991) 23–32.

[43] A. Bookstein, S.T. Klein, T. Raita, *ACM Trans. on Information Systems* **15** (1997) 254–290.

[44] Y. Choueka, A.S. Fraenkel, S.T. Klein, E. Segal, *Proc. 9-th ACM-SIGIR Conf.,* Pisa; ACM, Baltimore, MD (1986) 88–96.

[45] E.J. Schuegraf, *Inf. Proc. and Management* **12** (1976) 377–384.

[46] J. Teuhola, *Inf. Proc. Letters* **7** (1978) 308–311.

[47] M. Jakobsson, *Inf. Proc. Letters* **7** (1978) 304–307.

[48] H. Wedekind, T. Härder, *Datenbanksysteme II,* B.-I. Wissenschaftsverlag, Mannheim (1976).

[49] O. Vallarino, *SIGPLAN Notices, Special Issue* Vol. **II** (1976) 108–114.

[50] M. Jakobsson, *Inf. Proc. Letters* **14** (1982) 147–149.

[51] A.S. Fraenkel, *Amer. Math. Monthly* **92** (1985) 105–114.

[52] R.S. Boyer, J.S. Moore, *Communications of the ACM* **20** (1977) 762–772.

[53] C. Faloutsos, S. Christodulakis, *ACM Trans. on Office Inf. Systems* **2** (1984) 267–288.