A STUDY OF EFFICIENT PARSING IN LZ ADAPTIVE DICTIONARY COMPRESSION

By

LIN KE

Bachelor of Science
Zhongshan University
Guangdong, P.R. China
1986

Master of Science
Zhongshan University
Guangdong, P.R. China
1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1997

A STUDY OF EFFICIENT PARSING

IN LZ ADAPTIVE DICTIONARY

COMPRESSION

Thesis Approved:

J Chandler
Thesis Advisor

H. Lu

Thomas C. Collins
Dean of the Graduate College

PREFACE

The purpose of this research was to study three commonly known efficient parsing problems of LZ adaptive dictionary compression schemes. They are the efficiency of finding the longest match between the look-ahead buffer and the text window, coding redundancy and parsing strategies. We introduced an AVL tree data structure to the original LZSS variant and the Knuth-Moore-Pratt string matching algorithm to the LZ77 variant, and compared their performances. We also tried to modify the one-bit flag fixed-length coding method of LZSS to a two-bit flag variable length coding method and investigated the effort. Finally, we discussed a newly presented Non-Greedy parsing strategy .

Acknowledgments and thanks go to my thesis advisor Professor John Chandler for his great help, guidance, and patience during the entire work. My thanks also go to Dr. Huizhu Lu and Dr. K.M. George for their helpful suggestions to my research. I also wish to give thanks to Dr. Kathleen Kaplan who spent a lot of time to answer my questions on string matching and generously lent me all the related papers and books.

I am very thankful for the love and encouragement from my families, my parents, my mother-in-law, my younger brother, especially my wife Kaiping Deng. I cannot imagine, without all the help from these kind people, I could have finished my research on time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Data compression seeks to reduce the number of bits used to store or transmit information. This field is so interesting that it has been extensively studied for more than 40 years and has grown in the last 25 years to a point where taking a comprehensive look at every variety of compression scheme is simply not possible. In general, the techniques can be categorized into two major families: lossy and lossless. The former is usually applied to image and digitized speech compression where slight degradation of the information is acceptable, and the latter to text compression where complete accuracy is important. Approaches to text compression can be divided into ad hoc, statistical and dictionary techniques [Williams90] [Bell90] [Storer88], and they can be further characterized as static, semi-adaptive and adaptive methods, respectively (ad hoc techniques only use static methods), based on what kind of modeling they use. Statistical models generally encode a single symbol at a time; reading it in, calculating a probability, then outputting a single code. This kind of method can get fairly good compression (depending on how precisely the probability is calculated) but they are commonly slow. A dictionary-based scheme, on the other hand, uses a different concept. It reads in input data and looks for groups of symbols that appear in a dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression ratio. Dictionary-based schemes use relatively simple coding methods. They usually work fast and are easily implemented in

hardware. In addition, it has been proved that every dictionary scheme has an equivalent statistical scheme which achieves exactly the same compression [Bell90].

Until 1980, most general compression schemes used statistical modeling. But in 1977 and 1978, Jacob Ziv and Abraham Lempel described a pair of compression methods using an adaptive dictionary [Ziv77][Ziv78]. The appearance of these two algorithms attracted much study that has resulted in many variants. We may call these variants two families of LZ algorithms.

The main two factors that differ between variants of LZ coding are whether there is a limit to how far back a pointer can reach, and which sub-string within this limit may be the target of a pointer. The reach of a pointer into earlier text may be unrestricted (a growing window), or it may be restricted to a fixed-size window of the previous characters. The choice of sub-strings can be either be unrestricted or limited to a set of phrases, or limited to a set of phrases chosen according to some heuristic.

1.1 PROBLEM STATEMENT

LZ families use adaptive dictionary schemes to compress text data. Because dictionary coding achieves compression by replacing groups of consecutive characters (phrases) each time with indexes into the dictionary, instead of encoding one character each time as a statistical method does, its speed is relatively faster. But one of the most time-consuming bottle-necks of any LZ77 compression scheme lies in the speed of finding the longest match (prefix) between two strings. In the thesis, we will first focus on this efficiency problem.

The other research point is the size of each output coding. It is obvious that the smaller size the better, as long as the output coding contains the same (compact) information.

Another research point to be investigated here is the parsing strategies problem. Finding a longest match in itself is a greedy parsing strategy. Most variants use this strategy, but there are still some variants that insist on using "non-greedy" parsing. There exists some argument between these two parsing strategies.

Intuitively, the larger the dictionary referenced, the better the compression result that will be achieved, but it needs much more memory and careful maintenance. As the computer memory becomes more and more inexpensive, higher speed and better compression ratio have gained more concern.

In this thesis, we are going to probe into the above three problems and try some new approaches. We will use two well-known variants of LZ families: LZSS and LZW for the performance comparison.

1.2 ORGANIZATION OF THIS THESIS

The thesis will be organized as follows: Chapter I explains the motivation for the research and the thesis structure. Chapter II introduces some fundamental concepts and computational theories for data compression, including an overview on text compression techniques, especially on LZ families. Chapter III will discuss the string matching problems by investigating two effective algorithms in finding the longest match. In Chapter IV we will do some performance comparisons of different string matching schemes and

give a complete analysis on how we improved the original LZSS variant. Chapter V will emphasize the coding redundancy problem. In this section, we try to employ one extra bit in the original LZSS to examine the effect on the compression ratio. Chapter VI discusses "Greedy" and "Non-Greedy" parsing problems by analyzing the time complexity of one newly presented LZW version using "Non-Greedy" parsing schemes. The last chapter summarizes all our work and suggests further study.

CHAPTER II

AN OVERVIEW ON TEXT COMPRESSION TECHNIQUES

2.1 COMPUTATIONAL THEORIES FOR DATA COMPRESSION

There are two areas of computational theory that deal with representation of information: *Coding Theory* and *Information Theory*. Coding Theory deals with reliability in transmitting information, and Information Theory deals with efficiency in transmitting information.

In Information Theory, *Entropy* is a term used as a measure of the quantity of information. The word entropy was borrowed from thermodynamics, and it has a similar meaning. The higher the entropy of a message, the more information it contains. The entropy of a symbol is defined as the negative logarithm of its probability. To determine the information content of a message in bits, we express the entropy using the base 2 logarithm:

$$\text{Number of bits} = - \text{Log}_2 \text{ (probability)}$$

This means that more likely messages, having greater probabilities, contain less information. In other words, the more surprising the message, the more information it contains. The entropy of an entire message is simply the sum of the entropy of all individual symbols.

We should mention here that, unlike the thermodynamic measure of entropy, we can use no absolute number for the information content of a given message. The

probability for a given symbol is actually the probability for a given model, not an absolute number. If we change the model, the probability will change with it.

Based on the idea that entropy can determine how many bits of information are actually presented in a message, we can use a technique to reduce the information redundancy. It is *data compression*. This technique is best described using the following formula:

Data Compression = Modeling + Coding

That is, a data compression process can be viewed as a combination of two separate processes: modeling and coding. A *model* is simply a collection of data and rules used to process input symbols and determine which code(s) to output, so *modeling* is the process of applying these rules. Symbols that are to be stored or manipulated by a computer are converted to codes. This process is referred to as *coding*.

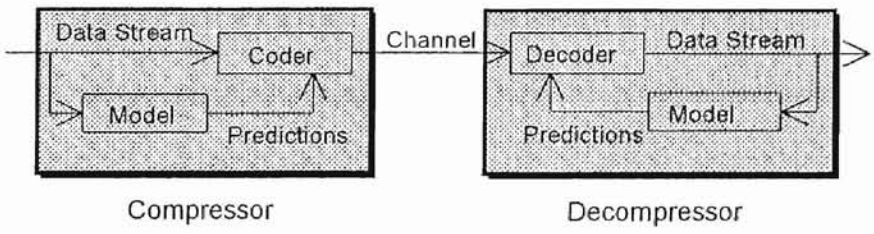


Figure 2-1 Data Compression = Modeling + Coding

The task of finding a suitable model for a given text is an extremely important problem in compression. There are three ways that the encoder and decoder can maintain

the same model: *static*, *semi-adaptive*, and *adaptive*. In *static* modeling, the encoder and decoder agree on a fixed model (i.e. a code book) regardless of the text to be encoded. During the process of compression and decompression, the model does not change. This kind of modeling is simple, easy to implement, but not highly efficient because both sides must have a large "general purpose code book." A *semi-adaptive* scheme is thus introduced to create an even better suited "code book." Before the encoder encodes a message, the encoder reads the message, writes down the most frequently used words, and uses them to make up a code book specifically for that message. In the second pass, it encodes the message by checking its newly created code book. But this kind of scheme must manage to pass that code book to its receiver, usually by embedding a code table into the compressed file. This is generally better than a static method because it uses a better suited and relatively smaller dictionary. The semi-adaptive method could be impractical in network communication because of the two-pass nature of parsing. In *adaptive* modeling, on the other hand, the modeling is continuously self-modified based on the "knowledge" it just "learned" from the text transmitted. This kind of method only needs one pass, which is very useful especially in a network environment where the end of a message ensemble is not predictable.

We can say that a static technique will always yield better compression than an adaptive technique on the data for which the static technique has been tuned. This is because the adaptive technique must spend time learning what the static technique already "knows". But static models may exhibit an unbounded inefficiency if they are fed unsuitable data. Thus, the only advantage that static models have over dynamic models is a

constant-order compression advantage for the particular kind of data for which they were tuned.

2.2 ADAPTIVE DICTIONARY COMPRESSION

Approaches to text compression can be divided into two classes: *statistical* and *dictionary*. In statistical coding each symbol is assigned a code based on the probability that it will occur. Highly probable symbols get short codes, and vice versa. For instance, the statistical Hoffman coding uses this method. In dictionary coding groups of consecutive characters, or "phrases," are replaced by a shorter code. The phrase represented by the code can be found by looking it up in a "dictionary." It is only recently that it has been shown that most practical dictionary coding schemes can be outperformed by a related statistical coding scheme.

The Dictionary approach became popular for several reasons. First, it is intuitive. Second, the size of the indexes to the dictionary can be chosen to align with machine words, expediting an efficient implementation. Third, it can achieve good compression with higher speed and lower memory requirement.

Like statistical approaches, dictionary approaches also use static, semi-adaptive or adaptive modeling. But they have to find some way to pass the dictionary from the transmitter to the receiver if necessary. Static techniques do not need to do this since both sides already have a "ready-to-use" dictionary; semi-adaptive techniques need to transmit the dictionary before transmitting the message; adaptive techniques, like static methods, do not need to transmit the dictionary. Instead, the transmitter and receiver both build the

same dictionary incrementally, adding to it as each instance (or group of instances) is transmitted. At each point during the coding, the current dictionary is used to transmit the next portion of the message.

2.2.1 Adaptive Dictionary Encoders: Zip-Lempel Coding

Almost all practical adaptive dictionary encoders are encompassed by a family of algorithms derived from Ziv and Lempel's work [Ziv77] [Ziv78]. This family is called Ziv-Lempel coding, abbreviated as LZ coding.

Ziv-Lempel coding refers to two distinct but related coding techniques first presented by Ziv and Lempel in two papers published in 1977 and 1978. The fundamental idea behind LZ algorithms is that sub-strings of the message are replaced by a reference (e.g. an <offset, length> tuple) to a sub-string in an earlier part of the message.

The first compression algorithm described by Ziv and Lempel is commonly referred to as LZ77. It is relatively simple. The dictionary consists of all the strings in a window into the previously read input stream. A file-compression program, for example, could use a 4K-byte window as a dictionary. While new groups of symbols are being read in, the algorithm looks for matches with strings found in the previous 4K bytes of data already read in. Any matches are encoded as pointers sent to the output stream.

LZ77 and its variants make attractive compression algorithms. Maintaining the model is simple; encoding the output is simple; and programs that work very quickly can be written using LZ77. Programs such as PKZIP and LHarc use variants of the LZ77 algorithm, and they have proven very popular.

The LZ78 program takes a different approach to building and maintaining the dictionary. Instead of having a limited-size window into the preceding text, LZ78 builds its dictionary out of all the previously seen symbols in the input text. But instead of having carte blanche access to all the symbol strings in the preceding text, a dictionary of strings is built, a single character at a time. This incremental procedure works very well at isolating frequently-used strings and adding them to the table. Unlike LZ77 methods, strings in LZ78 can be extremely long, which allows for high-compression ratios.

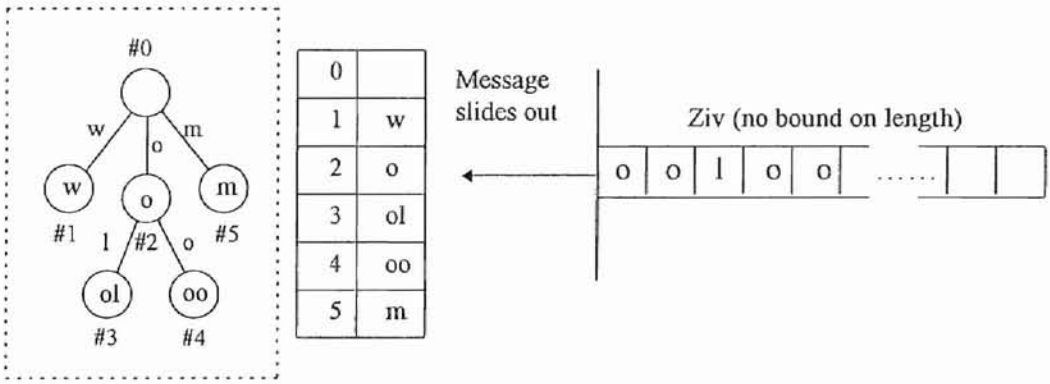


Figure 2-2 The LZ78 Algorithm

The LZ78 algorithm builds a dictionary of phrases and repeatedly transmits the number of the longest dictionary entry matching the Ziv, followed by one instance after that. After each phrase is transmitted, a new phrase is added to the dictionary. The new phrase is the phrase transmitted plus the following instance.

The dictionary can be visually described (orginazed) as a multi-way tree (trie). At each step, the algorithm transmits the number of the tip of the longest matching branch. The new phrase is then added by appending a new leaf to the tip node.

The above diagram depicts the processing of the string wooloomooloo using LZ78 algorithm.

2.2.2 LZ Variants

Bell, Cleary and Witten [Bell90] identified four LZ77 variants and six LZ78 variants. They are listed in Table 2-1.

In this chapter, we are going to focus our discussion on two typical variants of LZ77 and LZ78, respectively. They are LZSS and LZW.

LZSS

LZSS improved the original LZ77 in two approaches: using a binary tree to maintain the dictionary, which significantly speeds up the procedure of finding the longest match; eliminating the third output element in the original output triple to achieve a better compression ratio.

Based on the text window size and the look-ahead buffer size it is using during encoding, LZSS employs a "break-point" to determine whether to output single character(s) or a pointer tuple. For example, if the longest match is two unencoded characters, then the output will require 16 bits. But if we use a pointer tuple for the output, it needs 17 bits, totally, and we should simply output the two characters instead of outputting the pointer tuple. This break-point changes with the text window size and look-ahead buffer size defined.

An extra bit is added as a flag to each pointer tuple or character to distinguish between them. For instance, bit 0 denotes the following encoded content is a character. When decoding, the decoder will read 8 bits after the bit 0; bit 1 denotes the following encoded content is a pointer tuple, the decoder will then read a fixed number of bits for the match location and a fixed number of bits for the match length.

Table 2-1 Summary Of Principal LZ Variations [Bell90]

LZ77	Ziv and Lempel (1977)	Pointers and characters alternate Pointers indicate a substring in the previous N characters
LZR	Rodeh et al. (1981)	Pointers and characters alternate Pointers indicate a substring anywhere in the previous characters

LZSS	Bell (1986)	Pointers and characters and distinguished by a flag bit Pointers indicate a substring in the previous N characters
LZB	Bell(1987)	Same as LZSS, except a different coding is used for pointers
LZH	Brent(1987)	Same as LZSS, except Huffman coding is used for pointers on a second pass
LZW	Welch (1984)	The output contains pointers only Pointers indicate a previously parsed substring Pointers are of fixed size
LZC	Thomas et al. (1985)	The output contains pointers only Pointers indicate a previously parsed substring
LZT	Tischer (1987)	Same as LZC but with phrases in a LRU list
LZMW	Miller and Wegman (1984)	Same as LZT but phrases are built by concatenating the previous two phrases
LZJ	jakobsson (1985)	The output contains pointers only Pointers indicate a substring anywhere in the previous characters
LZFG	Fiala and Greene (1989)	Pointers select a node in a trie Strings in the trie are from a sliding window

LZW

The transition from LZ78 to LZW parallels that from LZ77 to LZSS. The output code of the original LZ78 consists of an index and a character. The index is a pointer to the dictionary, while the character is the first character that mismatches the dictionary. For example, for a coming character stream "hekkoss .. ..", the longest match we find in the dictionary is "hekk" at index 354, then the output will be <354, "o">.

Like LZSS, LZW mainly improved the original LZ78 in two approaches: The initialized dictionary is preloaded with 256 characters (the alphabet of ASCII) instead of the original one containing only an empty string, eliminating the last character of each new phrase in the original output code because it can be encoded as the first character of the

next phrase. These two approaches guarantee that the output code is always a dictionary entry.

2.3 PROBLEMS WITH LZ FAMILIES

There are three kinds of problems that will be encountered by the original LZ77 and LZ78 algorithms. They are string matching, output code redundancy and parsing strategies.

2.3.1 String Matching Problem

This is clearly a major performance bottleneck problem in the LZ77 approach. When encoding, it has to perform string comparison against the look-ahead buffer for every position in the text window. As we want to improve compression performance by increasing the text window size and thus the dictionary, this performance bottleneck only gets worse.

LZ78, on the other hand, uses a different way to find the longest match. It needs to walk through the dictionary tree, so the traversing efficiency is critical.

2.3.2 Coding Redundancy

To increase the compression rate, we can increase both the size of the text window and the size of the look-ahead buffer. But at the same time, we need to use more bits to represent those indexes in the output tuples, which may, on the contrary, decrease the compression rate. This problem happens both in LZ77 and LZ78.

2.3.3 Parsing strategies

Once a dictionary has been chosen, there is more than one way to choose which phrases in the input text will be replaced by indexes to the dictionary. The task of splitting the text into phrases for coding is called *parsing*. The most practical approach is *greedy* parsing, where at each step the encoder searches for the longest string (prefix) to encode them. For example, if a dictionary M={a,b,ba,bb,abb}, and coding C(a)=00, C(b)=010, C(ba)=0110, C(bb)=0111, and C(abb)=1, then "babb" is coded greedily in 8 bits as C(ba).C(ab)=0110.0111.

Unfortunately, "greedy" parsing is not necessary optimal. In the above example, "babb" can also be coded in only 4 bits as C(b).C(abb)=010.1. However, determining a "complete" optimal parsing can be difficult in practice, because there is no limit to how far ahead the encoder may have to look.

CHAPTER III

FINDING THE LONGEST MATCH

3.1 AN OVERVIEW OF DIFFERENT SCHEMES

Both LZ77 and LZ78 data compression techniques need to find the longest (prefix) match between the string in the look-ahead buffer and the one in the dictionary, which is the most time-consuming process during the compression.

Most string matching algorithms deal with finding an exact match. Some of them have been proved very practical, like the Boyer-Moore algorithm and its variants. But the algorithms for approximate match can hardly be expected to be as efficient because of two reasons: first, we do not know the match length in advance; second, we have to keep the knowledge of each previous (prefix) match and thus guarantee that each shift will not miss the potential longest match.

Some known data structures that can be used to find the longest match are listed below [Bell90] (suppose N is the number of characters in the text window).

3.1.1 Unsorted List

This is a trivial way to find a longest match by performing a linear search of the text in the window. Since the window is fixed-size, the time this algorithm takes is independent of the text size; in addition, it requires relatively little memory. However, the approach is slow since a window of N characters requires at least N character comparisons. For example, finding the longest match in the following unsorted list (Figure

3-1) [Bell90] is actually doing a string pattern searching between "bcbacbabab" and "babc".

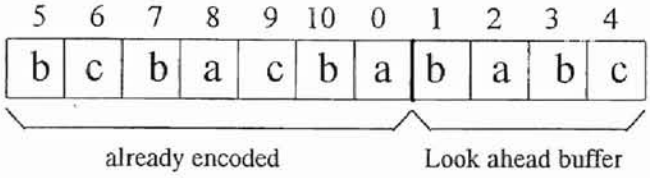


Figure 3-1 Using unsorted list to find the longest match [Bell90]

Note: in Figure 3-1, number serial 5,6,...3,4 is a modulo indexing of N=11, because in LZ77 schemes, we need to shift the characters in the text window. This is used to simplify the process by overwriting the characters to be shifted.

3.1.2 Sorted List

This is a more sophisticated data structure than an unsorted list. Suppose that a string l is inserted at its correct position in a sorted list, between the strings x_a and x_b. Then the longest match for l in the list will be a prefix of either x_a or x_b. Figure 3-2 [Bell90] shows a sorted list that corresponds to the window of Figure 3-1. Obviously, if l= "babc", the longest (prefix) match is "bab", the prefix of x_a. Finding the point of insertion can be achieved by a binary search, using about log$N$ substring comparisons, but insertion and deletion require O(N) operations.

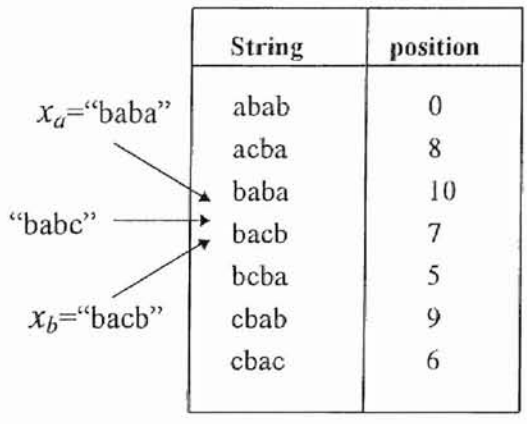


Figure 3-2 Using sorted list to find the longest match [Bell90]

3.1.3 Binary Tree

A successful variant of LZ77, LZSS, uses a binary search tree to find the longest match to make the insertion and deletion of sub-strings more efficient than for the sorted list. Figure 3-3 [Bell90] shows a binary tree containing the strings of the previous example. When the string "babc" is inserted, the two longest match candidates will be on the path from the root to the point (node) of insertion.

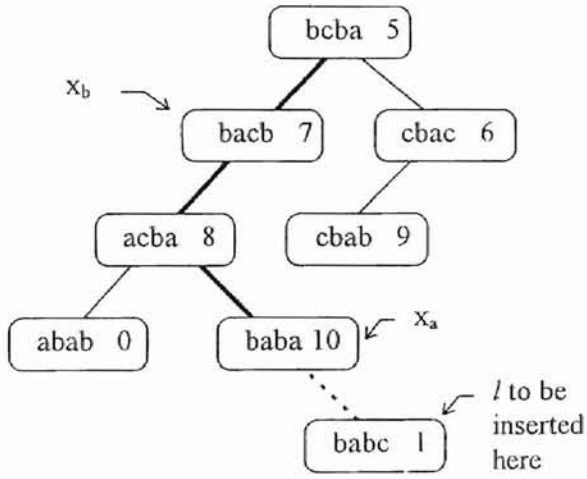


Figure 3-3 Using a binary tree to find the longest match [Bell90].

We need to further explain the rules for finding the longest match in a binary tree as follows:

"If x_a is the parent node of l, then x_b is the node where the insertion path last turned left; otherwise, x_a is the node where the insertion path last turned right."

For instance, in Figure 3-3 if l is "babc", then x_a is "baba" and x_b is "bacb" (the last turn left node), because both "babc" and "bacb" are the two closest nodes to "babc".

3.1.4 Trie

A trie is a multiway tree with a path from the root to a unique node for each string represented in the tree. Figure 3-4 [Bell90] gives an example of a trie indexing a window. The most important feature of this data structure is that only the unique prefix of each string is stored, as the suffix can be determined by looking up the string in the window. A longest match is found by following down the tree until no match is found, or the path ends at a leaf. And the longest match can be found at the same time as inserting a new string, as with the binary tree algorithm.

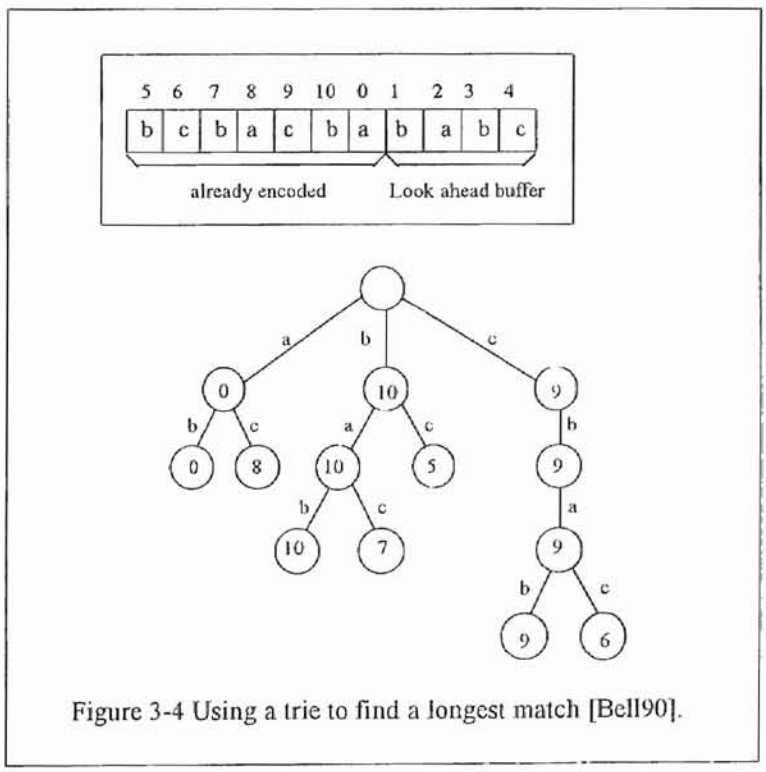


Figure 3-4 Using a trie to find a longest match [Bell90].

There are many optional ways in designing the nodes' data structure of a trie. For example, each node may contain 256 possible branches (the total number of ASCII characters). This is the fastest and simplest way but will waste much memory space

because each node uses only a portion of those "possibilities". The other approach is using a linked-list at each node, with one item for each possible branch. This uses memory economically, but the approach can be slower due to the time taken to search the list. A commonly optimal technique used is using a single hash table with an entry for each node. To determine the location of the child of the node at location n in the table for input character c, the hash function is supplied with both n and c. This algorithm is used successfully in the programs "compress" (LZC) and LZW.

3.1.5 Hash Table

Hash tables are normally associated with exact matches. It is possible to use them to find longest matches if all prefixes of each substring are inserted in the hash table. Then to find the longest match of the string x, we first test whether x[1] is in the table, then x[1..2], then x[1..3], and so on. If x[1..i+1] is the first such string not found, the longest match must be x[1..i]. This approach is really equivalent to the implementation of a trie using a signal hash table, as mentioned above.

3.2 CASE STUDY 1: APPLYING KMP ALGORITHM TO DATA COMPRESSION

Rodeh et al. [Rodeh81] pointed out that a straight-forward implementation of the Ziv-Lempel algorithm takes $O(n^2)$ time to process a string of length n. Hashing was proposed by Welch [Welch84] to achieve $O(n)$ processing time. The Knuth-Morris-Pratt algorithm is one of the best linear-time string matching algorithms. The algorithm achieves

a $\Theta(n + m)$ running time. It does the pattern matching using an auxiliary function $\pi[1..m]$ (prefix function) precomputed from the pattern in time $O(m)$.

Even though the original algorithm is used to find an exact match, we can modify it easily to adapt for finding the longest (prefix) match without pain. For a comparison purposes, we first implement an LZ compression program using the KMP string matching scheme, named as LZKMP.

3.2.1 The Prefix Function For A Pattern Of KMP Algorithm

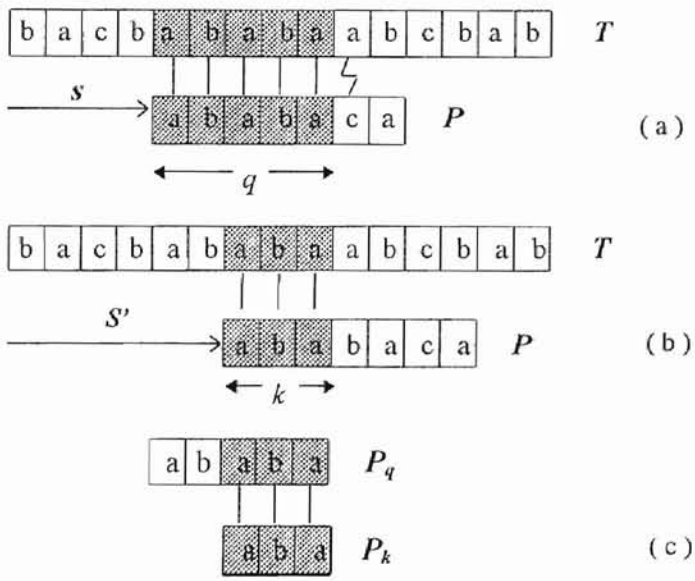


Figure 3-5 An example of how KMP algorithm works [Corman90].

The prefix function π. (a) The pattern P = ababaca is aligned with a text T so the first q=5 characters match. Matching characters, shown shaded, are connected by vertical lines. (b) Using only our knowledge of the 5 matched characters, we can deduce that a shift of s+1 is invalid, but that a shift of s'=s+2 is consistent with everything we know about the text and therefore is potentially valid. (c) The useful information for such deductions can be precomputed by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a suffix of π_5 is π_3. This information is precomputed and represented in the array π, so that $\pi[5]$=3. Given that q characters have matched successfully at shifts, the next potential valid shift is at s'=s+(q-$\pi[q]$).

The prefix function of the KMP algorithm for a pattern encapsulates the knowledge about how the pattern matches against shifts of itself. This information can be used to avoid testing useless shifts in the straight-forward pattern-matching algorithm.

We formalize the pre-computation as follows. Given a pattern $P[1..m]$, the prefix function for the pattern P is the function π: $\{1,2,\ldots,m\} \rightarrow \{0,1,\ldots,m-1\}$ such that $\pi[q] = \max\{ k : k < q \text{ and } P_k \supset P_q \}$. That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

3.2.2 An Example Of How KMP Algorithm Works

Figure 3-5 [Corman90] shows an example of two string matching using the KMP algorithm.

3.2.3 Adapt KMP for finding the longest match

The original KMP matching algorithm is given in pseudo-code below:

KMP-Matcher(*T, P* **)**

```
1     n ← length[ T ]
2     m ← length[ P ]
3     π ← Compute-Prefix-Function(P)
4     q ← 0
5     for i ← 1 to n
6           do while q > 0 and P[ q+1 ] ≠ T[ i ]
7                 do q ← π[ q ]
8           if P[ q+1 ] = T[ i ]
9                 then q ← q+1
10          if q = m
11                then print "Pattern occurs with shift" i-m
12                      q ← π[ q ]
```

```
Compute-Prefix-Function( P )
1     m ← length[ P ]
2     π[ 1 ] ← 0
3     k ← 0
4     for q ← 2 to m
5           do while k > 0 and P[ k+1 ] ≠ P[ q ]
6                 do k ← π[ q ]
7              if P[ k+1 ] = P[ q ]
8                 then k ← k + 1
9              π[ q ] ← k
10    return π
```

From the above algorithm we know that KMP guarantees that each step has q prefix characters that match. Each shift for the next step is computed by $\pi[\ q\ ]$ ($s' = s + (\ q - \pi[q]\ )$. So if we trace the largest q and its position, we can adapt KMP to finding the longest prefix match.

3.2.4 Applying the KMP To A Data Compression Program

Applying the KMP algorithm to a data compression program is not a very difficult task. We only need to use a text window (an array containing both the previously encoded text and the look-ahead buffer) as the data structure.

unsigned char window[WINDOW_SIZE]

It is a simple data structure. But how can we implement the "Sliding Window" on this character array?

A "Sliding Window" is just used, for conceptual convenience, to discuss the algorithm. We imagine it as though it were truly sliding "across" the text, progressing from the end of the buffer to the front as the encoding process was executed. But in the

implementation, we cannot actually "slide" the content of a text window (a very long string) because moving several thousand bytes of memory will waste much more time than finding the longest match itself.

A workable solution could be, as mentioned in Figure 3-1 note part: We employ a sliding index or pointer into a fixed buffer (text window). Instead of moving the phrases toward the front of the window, a sliding pointer would keep the text in the same place in the window and move the start and end pointers along the buffer as text is encoded. This idea is illustrated as follow.

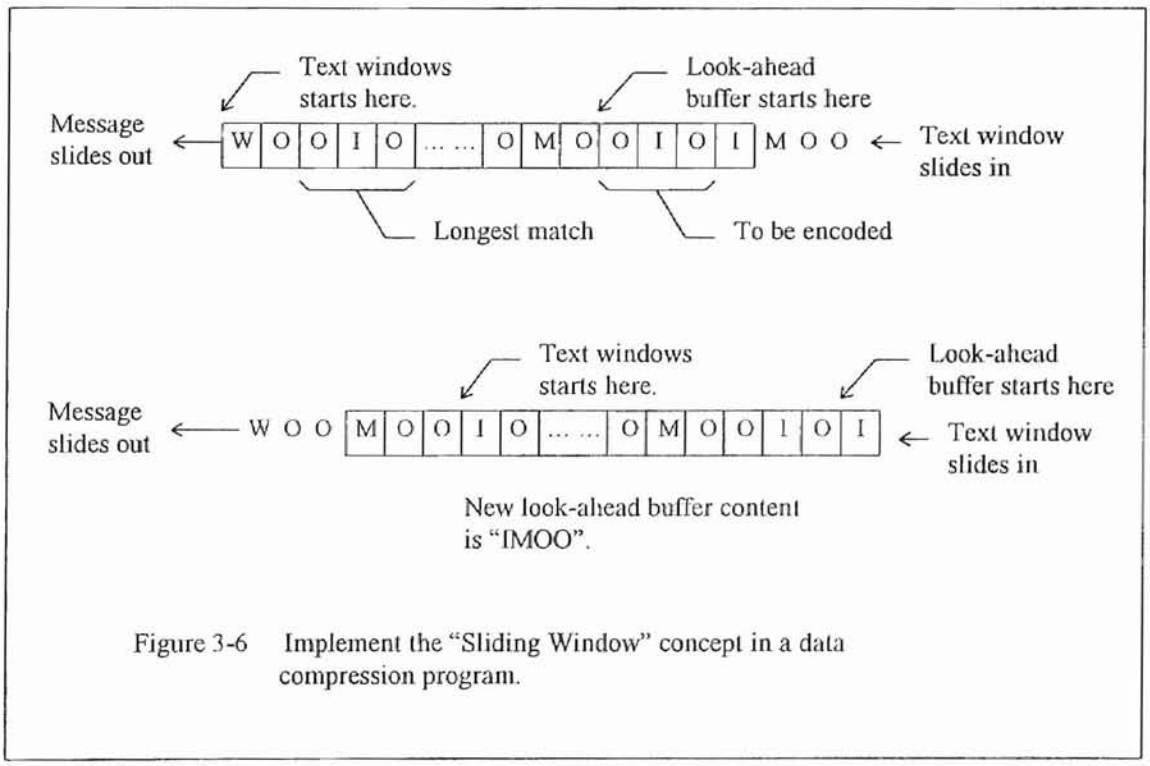


Figure 3-6 Implement the "Sliding Window" concept in a data compression program.

The output index will be a module index into the window instead of a normal index. This solution will make the program more complicated, but it will pay for itself in savings on calls of memmove().

3.3 CASE STUDY 2: AN IMPROVEMENT APPROACH TO THE LZSS

3.3.1 Using An Binary Tree To Find The Longest Match

Timothy C. Bell introduced a binary tree in the LZ77 variant, LZSS, to speed up the longest match location process in his paper "Better OPM/L Text Compression" [Bell86]. His algorithm also works where a longest string match is required without a limitation in compression area.

3.3.2 Data Structure Used In The Original LZSS

As mentioned in 3.1.3, this scheme needs to employ an extra data structure to maintain the binary tree:

```
struct {
        int     parent;
        int     smaller_child;
        int     larger_child;
} tree[ WINDOW_SIZE+1 ];
```

Notice here that in the array tree[], we declare one more node. This is a special node--the root node. In the program, the position tree[WINDOW_SIZE] will never be used by any other nodes except the tree root.

For every phrase in the window, a corresponding structure element defines the position that the phrase occupies in the tree. Each phrase has a parent and two children pointers. Since this is a binary search tree, the two child nodes are defined as "smaller" and "larger" children. Every phrase that resides under the smaller_child node must be smaller

than the phrase defined by the current node, and every phrase under the larger_child node must be larger than the phrase defined by the current node. The terms "larger" and "smaller" refer to where the phrases fall in the collating sequence used by the compression program. In this particular program, one phrase is "larger" or "smaller" in the same sense as that used by the standard library strcmp() function.

The root node does not have a phrase of its own, as do all other nodes in the tree. It also does not have smaller and larger children like other nodes. Instead, it has the index of a larger_child only, and this index points to the root node of the tree.

The other special node is at position zero in the text window. Since the content to be compared with the look-ahead buffer is one character less than the complete text window, we actually use (at most) WINDOW_SIZE-1 nodes. We thus only use index 1 to WINDOW_SIZE-1 and reserve position zero and WINDOW_SIZE for special uses. If we want to mark an "unused" node or "deleted" node, we set their parents as zero. Similarly, we can also mark its children as "unused" by setting their value zero.

3.3.3 Implementation Of LZSS Using Binary Tree Algorithm

During the encoding of the LZSS, the tree is continuously updated as the content of the window changes. Each time the window moves along a character, the character s(i) leaves the window. The tree is searched for the associated phrase x_i, and if it is found, it is deleted. Also, one character s(j) enters the window, and the phrase in a new look-ahead buffer x_{j-F+1} is inserted in the tree. As we already described in 3.1.3, finding a longest (prefix) match of a string l should first find its two "closest" nodes x_a and x_b and do

comparison with them. We also know that these two nodes must be on the searching path for an insertion, so the insertion routine will also take care of the task of finding the longest match. This makes the program more efficient.

3.4 THE AVL TREE VERSION FOR THE LZSS-LZAVL

In the "Analysis of Running Time" section of T.C. Bell's paper [Bell86], he wrote:

"Each insertion requires a probe into the tree of O(LogN) string comparisons for a reasonably balanced tree, but O(N) in the worst case. If worst case behavior must be avoided, a balanced tree, such as an AVL tree [Ziv77], could be used. On the average, each string comparison requires only a couple of character comparisons, but in the worst case F character comparisons could be required...."

where N is the size of text window, F is the size of look-ahead buffer.

The author suggested that an AVL tree can be used to improve the worst case of the LZSS. But as far as we know, no one has ever tried this method. So the second step of our work is to implement this AVL tree LZSS variant and compare their performances.

Definition of an AVL tree: An AVL tree is a binary tree of *height-balanced* type if and only if it consists of a single external node, or the two subtrees T_l and T_r of the root satisfy

1. $| h(T_l) - h(T_r) | \leq 1$.
2. T_l and T_r are height-balanced.

3.4.1 Data Structure Used In LZAVL

For an AVL tree, we need to handle the height balance. In order to verify/restore the tree's height balance condition, we need to be able to test whether the element inserted or deleted has changed the relationship between the heights of the subtrees of a node so as to violate the height constraints. For this purpose, we will store a *balance scale code* in each node of a height-balanced tree. The balance scale code is one of the following:

> Means the left subtree of this node is taller (by one) than its right subtree.

= Means the two subtrees of this node have equal height.

< Means the right subtree of this node is taller (by one) than its left subtree.

Storing this height scale requires an extra two-bit field per node in the tree. But for programming convenience, we use a character (8-bit) to store this condition code. This won't affect the compression rate but uses a little bit more memory.

```
Struct {
        int     parent;
        int     smaller_child;
        int     larger_child;
        char    height_scale;
} tree[ WINDOW_SIZE+1 ];
```

We should mention here that many AVL tree algorithms use a recursive technique to implement the insertion and deletion routines. But for comparison purposes in this thesis, we don't use any recursive insertion or deletion for the AVL tree. Instead, we do the rebalancing process right after each insertion or deletion. In those cases, we can simply

separately calculate or measure the time consumed by these operations. By a careful selection, we find the following algorithm meets our need.

3.4.2 The Algorithm for Rebalancing After Insertion Or Deletion

Roughly speaking, the rebalancing process consists of retracing the path upward from the newly inserted node (or from the site of the deletion) to the root. Since we have "parent" pointer in each node, this can be done most efficiently. As the path is followed upward, we check for instances of the taller subtree growing taller (on an insertion) or the shorter subtree becoming shorter (on a deletion). When we find such an occurrence, we apply a local transformation to the tree at that point. In the case of an insertion it will turn out that applying the transformation at the first such occurrence will completely rebalance the tree. In the case of a deletion the transformations may need to be applied at many points along the way up to the root, in the worst cases.

It has been well known that *single-rotation* and *double-rotation* could be used to repair the newly created imbalance, which can be illustrated in Figure 3-7 [Reingold83]:

The insertion algorithm for the AVL tree version is thus as follows. Use a standard binary tree insertion algorithm to insert the new element into its proper place, setting its height scale code to = . Then, retrace the insertion path by pointer "parent" backward up the tree and correct the height scale codes until either the root is reached and its height scale code corrected, or we reach a point when no more height scale codes need to be corrected, or we reach a point at which a rotation or double rotation is necessary to rebalance the tree. More specifically, we follow this path backward, node by node, taking

actions as defined by the following rules, where *current* is the current node on the path, *son* is the node just before the current node on the path (that is, its son), and *grandson* is the node before *son* on the path (the grandson of *current*). Initially, *son* is the new element just inserted, *current* is its father, and *grandson* is **nil**:

1. If *current* has height_scale code =, change it to < if *son* = larger_child(*current*) and to > if *son* = smaller_child(*current*). In this case the subtree rooted at son grew taller by one unit, causing the subtree rooted at *current* to grow taller by one unit, so we continue up the path, unless current is the root, in which case we are done. To continue up the path we set *grandson* ← *son*, *son* ← *current*, and *current* ← parent(*current*).
2. If *current* has height_scale code > and *son* = larger_child(*current*) or *current* has height_scale < and *son* = smaller_child(*current*), change the height scale code of *current* to =, and the procedure terminates. In this case the shorter of the two subtrees of *current* has grown one unit taller, making the tree better balanced.
3. If *current* has height scale code > and *son* = smaller_child(*current*) or *current* has height scale code < and *son* = larger_child(*current*), then the taller of the two subtrees of *current* has become one unit taller, unbalancing the tree at *current*. A transformation is performed according to the following four cases :

	grandson = larger_child(*son*)	grandson = smaller_child(*son*)
son = larger_child(*current*)	Rotate around *current* using Figure 3-7 (a)	Double-rotate *around* current using Figure 3-7 (b)
son= smaller_child(*current*)	Double-rotate *around* current using the mirror image of Figure 3-7 (b)	Rotate around *current* using the mirror image of Figure 3-7 (a)

Table 3-1 The four cases of insertion of an AVL tree

The deletion process is more complex than insertion because it will not always be sufficient to apply a transformation only at the lowest point of imbalance; transformations may need to be applied at many levels between the site of the deletion and the root. To delete a node from a height-balanced tree, we proceed as for unconstrained trees: if the node is a leaf, just delete it; if it has one non-nil son, replace it with its son; if it has two non-nil sons, replace it by its inorder predecessor (successor), which will have a null larger (smaller) child. So the AVL tree deletion process only needs to consider the case of deleting a node leaf node or a node with only one child. This makes things workable. We now describe the deletion algorithm as follows:

First, we use a standard binary tree search algorithm to find the node to be deleted. If the node found is not a leaf node, call a routine to find its replacement node (this node must be a leaf node or a node with only one child). Delete the targeted node. Set *current* to be the parent of node deleted, and *son* to be the node deleted. Going up the path, doing rebalancing operations by following the rules below:

1. If *current* has height-scale code =, then shortening either subtree does not affect the height of the tree rooted at *current*. The height-scale code of *current* is changed to < if *son* = smaller_child(*current*) and to > if *son* = larger_child(*current*). The procedure then terminates.
2. If *current* has height-scale code < and *son* = larger_child(*current*) or *current* has height-scale code > and *son* = smaller_child*(current)*, the height-scale code of *current* is changed to =. The subtree rooted at *current* has become shorter by one unit, so we continue up the path, unless *current* is the root, in which

case we are done. To continue up the path we set *son* ← *current* and *current* ← parent*(current)*.

3. If *current* has height-scale code < and *son* = smaller_child*(current)*, then the height scale is violated at *current*. There are three subcases, depending on the height-scale code at larger_child*(current)*, the brother of *son*. The subcases are as given in Figure 3-8 [Reingold83].
4. If *current* has height-scale code > and *son* = larger_child*(current)*, then the height-scale is violated at *current*. There are three subcases, depending on the height-scale code at smaller_child*(current)*, the brother of *son*. The subcases are the mirror images of those given in Figure 3-8.

3.5 DESCRIPTION OF METHODS

In this chapter, for correct comparison purpose, we used straightforwardly an LZSS program from [Nelson96] and we programmed the LZAVL and LZKMP based on that program, because to compare the original LZSS and our LZAVL, the compression speed is our main concern. If the structures of two programs are totally different, we cannot tell whether the speed difference is from the difference of the two schemes or from the programming techniques. The two programs should be quite similar in structure except for some necessarily different parts-tree height balance maintenance, for example. So the LZAVL and the LZKMP programs only changed the related routines for maintaining the dictionary tree and finding the longest matches, respectively. This has guaranteed that the performance difference can be correctly measured and compared.

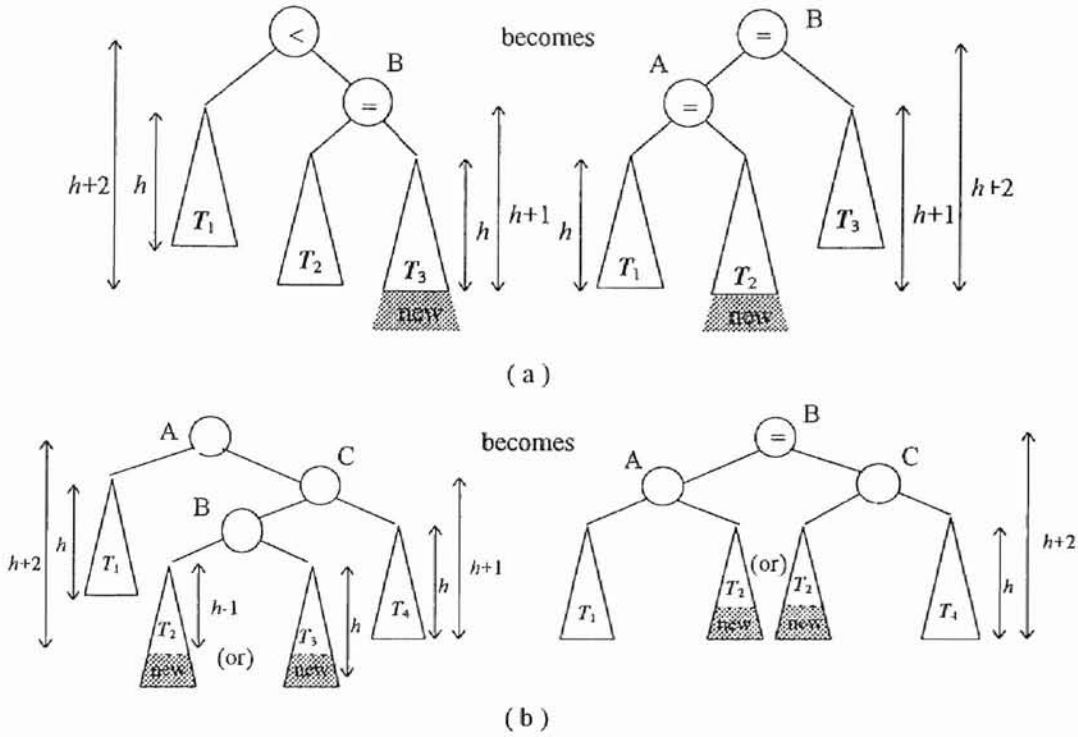
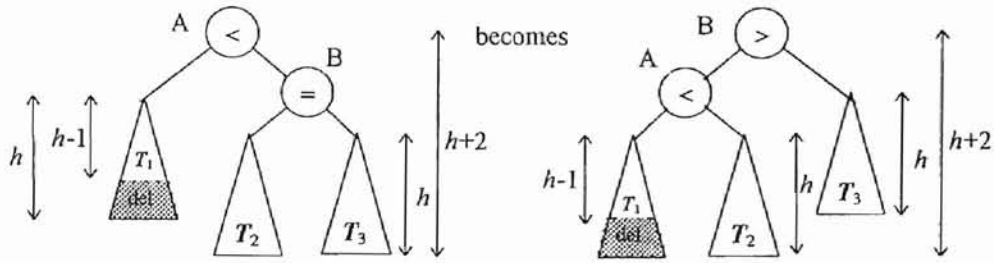


Figure 3-7 Two cases of rotations for insertion in an AVL tree [Reingold86]
The transformations used to rebalance a height-balanced tree after the insertion of a new element: (a) rotation around A, (b) double rotation around A. The height condition codes in A and C in the right-hand drawing of (b) depend on whether the new element is at the bottom of T2 or T3. Both T2 and T3 are empty when B is the new element Notice that in each transformation the inorder of the tree is unchanged and the height of the tree after the transformation is the same as the height of the tree before the insertion. In each case, there are corresponding mirror-image transformations.

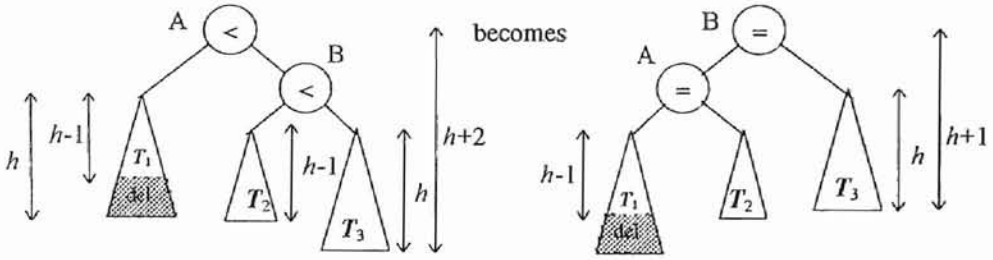
3.6 THE BOYER-MOORE ALGORITHM

The Boyer-Moore algorithm solves the pattern matching problem by repeatedly positioning the pattern over the text and attempting to match it. For each positioning that occurs, the algorithm starts matching the pattern against the text from the right end of the pattern. If no mismatch occurs, then the pattern has been found, otherwise the algorithm

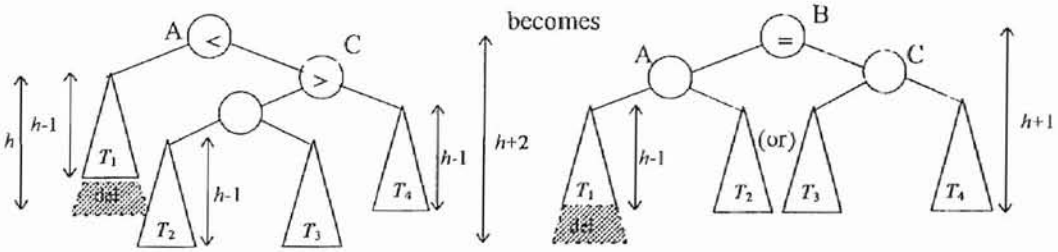
computes a shift that is an amount by which the pattern will be moved to the right before a new matching attempt is undertaken.



(a) Apply the rotation of Figure 3-7(a) to *current* and the procedure terminates, since the height-balance has been restored and the height of the tree after the transformation is the same as it was before the deletion.



(b) Apply the rotation of Figure 3-7(a) to *current*. If current is the root, the procedure terminates: otherwise continue up the path toward the root because the height of the subtree is now one less than it was before the deletion.



(c) Apply the double rotation of Figure 3-7 (b) to current. If current is the root, the procedure terminates; otherwise continue up the path toward the root because the height of the subtree is now on less than it was before the deletion. The height-scale codes of A and C are both = if that B was =. If B was <, then A is > and C is =. If B was >, then A is = and C is <.

Figure 3-8 Three cases of rotations for deletions in an AVL tree [Reingold86]

This algorithm works very fast because it incorporates two 'heuristics' that allow it to avoid much character comparison work. These heuristics are known as the "bad-character heuristic" and the "good-suffix heuristic"[Boyer77]. They can be viewed as operating independently in parallel. When a mismatch occurs, each heuristic proposes an amount by which the shift can safely be increased without missing a valid shift (i.e., missing a match).The algorithm will pick up the larger amount to use as the shift. We use the following example cited from the original paper [Boyer77] to illustrate how this algorithm works:

3.6.1 An Example of How the BM Algorithm Works

```
pattern:        AT-THAT
string:   ...   WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
```

1. Because "F" is known not to occur in the pattern (this can be known by a pre-computation), "Bad-heuristic" suggests a shift to the right of 7 characters (the length of the pattern).

```
pattern:              AT-THAT
string:   ...   WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
```

2. "-" occurs in the pattern. The "bad-character heuristic" suggests a right-shift of 4 characters.

```
pattern:                    AT-THAT
string:   ...   WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
```

3. "T" matches, but "L" doesn't occur in the pattern, the "Good-suffix heuristic" suggests a move of 3 character. The "bad-character heuristic" suggests a move of 5 characters. Then right-shift 5 characters.

pattern: AT-THAT
string: ... WHICH-FINALLY-HALTS .--AT-THAT-POINT ...

4. The "Good-suffix" "AT" suggests a shift of 5 characters. Mismatch occurs at "-" so that "bad-character" suggests a move of 3 characters. Then right-shift 5 characters.

pattern: AT-THAT
string: ... WHICH-FINALLY-HALTS .--AT-THAT-POINT ...

5. The match was found after only 14 references to string. A straightforward way needs to move past the first 22 characters of string.

The average running time is $O(n/m)$ (n = string length, m = pattern length), but the worst-case running time is somewhat like a straightforward comparison, $O(n*m)$. Efficient BM variants focus on how to improve the two heuristics to make the comparison progress faster or how to avoid the worst case.

3.6.2 Can the BM Algorithm Be Applied To Finding The Longest Match?

We have noticed that the Boyer-Moore algorithm has an attractive average running time of $O(n/m)$, which increases with the window size n getting larger, and decreases with the look-ahead buffer size m getting larger. A larger m and a larger n means we have a greater probability of finding a longer match which will cause better compression, but will not change the running time much. If the BM algorithm can be applied to data compression program, we can expect to get a good result.

But unfortunately, the answer is no. The BM algorithm can be used in the case of finding an exact match. Its two heuristics work together to find a maximum shift. But in

some cases, either of these heuristics can miss the longest match. An example is shown in Figure 3-9 [Corman90] as follows:

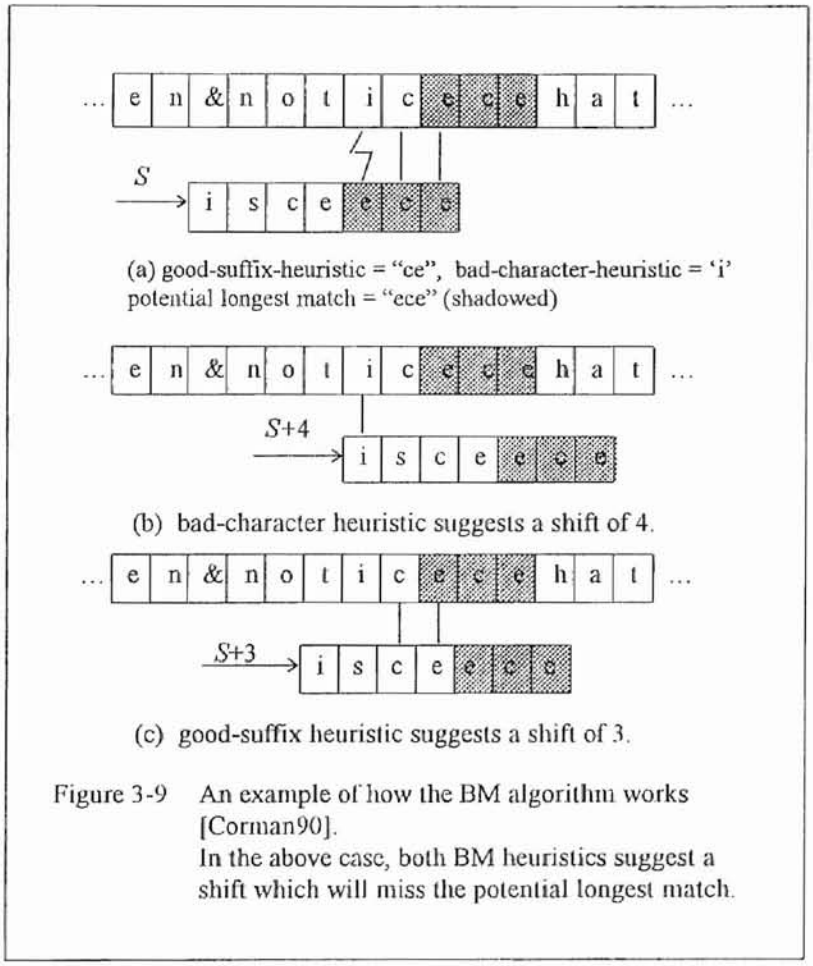


Figure 3-9 An example of how the BM algorithm works [Corman90].
In the above case, both BM heuristics suggest a shift which will miss the potential longest match.

CHAPTER IV

PERFORMANCE ANALYSIS AND COMPARISON

4.1 MEASURE OF PRACTICAL PERFORMANCE

The first practical measure for a text compression scheme is the amount of compression it achieves. However, there are other factors that are important in practice, namely, the amount of time and memory used by the encoding and decoding algorithms. Two main experimental measures are to be used in evaluating the performance of our schemes. They are compression ratio and compression speed (decompression speed will not be on the comparison list since the three schemes use the same decompression routine). We also briefly compare the memory used by each scheme.

There are several ways to measure the amount of compression achieved in an experiment, and each is widely used. If a text file of 1000 bytes in size is compressed to 250 bytes, this could be expressed as a compression ratio of 25% (or 0.25), or as a reduction of 75%, or as a factor of 4:1. In this thesis, we use the reduction rate definition

C.R. = [(input file size - output file size) / input file size] × 100 %

i.e. the compression ratio is the percentage by which the input file size has been reduced. The larger the compression ratio is, the better the compression. Because this ratio changes greatly with the type of input file being compressed, we also need to calculate the average compression ratio for each scheme.

Compression speeds are defined by the number of characters compressed per second. The time interval can be obtained by recording the two time points before and after calling the compression routine in the main program.

For further comparison purposes, we will measure the "average search length" of the LZSS and the LZAVL, respectively. We define the "search length" as the distance from the tree root to the target node. The distance between two adjacent nodes (parent and child) is defined as one. We can tell from these values how much the LZAVL has improved the tree height balance (its average search length in the tree should be shorter than that of LZSS).

The programs were not executed in a UNIX environment because in a multi-user system, we cannot record the running time correctly even though we only need to measure the relative speed. Instead, they were run in MS-DOS on a PC (Pentium 60, 8 Meg memory), which can guarantee that the relative speeds are correctly compared. All the programs were compiled using Visual C++ 1.52; they are 16-bit MS-DOS executable files. Obviously, two main factors that greatly affect the performance of an adaptive dictionary compression scheme are the size of the dictionary (for both families) and the size of the look-ahead buffer (for LZ77 family). The dictionary size is a main parameter. In general, the larger the text window the better the compression is (but the text window cannot be bigger than the text itself) .

4.2 CHOOSING TEST DATA

A text data compression program can be used to compress almost all kinds of data because of its lossless nature, but the compression ratio achieved depends on the type of data being compressed. For example, executable files tend to have little redundancy and get compression ratios of about 2:1 or less with any method. Windows 95's help files (*.hlp) are usually well "compacted" and likely to have very poor compression ratios. Databases (*.dbf) can have large amounts of redundancy, and some methods may get well over 10:1 compression ratios.

To evaluate the practical performance of the schemes described in this thesis, we collected 18 text files of 9 types. Each type has two files of different size. They are (1) C source codes (*.c), (2) VAX assembly source codes (*.vax), (3) Readme text files (readme1 and readme2), (4) Some chapters from an electronic books (book1 and book2, they are downloaded from internet and saved as text files.) (5) Papers in text format (paper1, paper2), (6) Visual Basic programs (*.vb), (7) Windows help files (*.hlp), (8) Executable files (*.exe), and (9) WORD 6.0 files (*.doc).

4.3 PERFORMANCE COMPARISON

Table 4-1 to 4-6 give the experimental results for the compression ratio and speed of three different compression schemes with different sizes of text window and look-ahead buffer. Because these three different schemes are using the same coding modeling, the compression ratios should be the same.

TABLE 4-1 COMPARISON OF RATIOS & SPEEDS

($N = 2^{\wedge}10$, $F = 2^{\wedge}4$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	56.1	56.1	55.9	42910	38065	12757
readme2	21500	47.2	47.2	47.0	39090	32575	10591
prog1.vb	25546	53.2	53.2	53.0	41878	38706	10824
prog2.vb	48480	53.5	53.5	53.4	40399	35386	11327
prog1.c	58303	63.2	63.2	63.0	40771	36668	14760
prog2.c	29207	63.2	63.2	62.9	41136	37931	14387
prog1.vax	6961	59.8	59.8	59.6	40947	40947	12656
prog2.vax	11453	66.4	66.4	66.1	30139	34706	14874
prog1.exe	25361	33.5	33.5	33.4	42268	38425	7708
prog2.exe	66672	42.7	42.7	42.6	39218	38098	8738
paper1	29672	45.4	45.4	45.3	44957	36185	10559
paper2	59055	47.0	47.0	46.9	41297	37141	10640
book1	93575	47.1	47.1	47.0	43726	36986	10855
book2	188613	45.2	45.2	45.1	42384	37722	10531
winhlp1.hlp	24099	26.8	26.8	26.7	43816	36513	6548
winhlp2.hlp	83833	14.1	14.1	14.0	39174	34785	5391
word1.doc	68642	46.1	46.1	46.0	39223	36904	10416
word2.doc	200192	56.9	56.9	56.7	40524	37630	12191
AVERAGE	61776	48.2	48.2	48.0	40770	36965	10875

TABLE 4-2 COMPARISON OF RATIOS & SPEEDS

($N = 2^{\wedge}11$, $F = 2^{\wedge}4$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	58.7	58.7	58.4	38065	36876	7814
readme2	21500	52.0	52.0	51.7	39090	39090	6739
prog1.vb	25546	53.3	53.3	53.1	35480	35480	6200
prog2.vb	48480	53.4	53.4	53.2	36727	35386	6395
prog1.c	58303	68.2	68.2	67.9	38106	36439	10104
prog2.c	29207	66.0	66.0	65.6	37931	35189	9014
prog1.vax	6961	61.7	61.7	61.6	40947	40947	7910
prog2.vax	11453	69.3	69.3	68.9	26029	34706	9959
prog1.exe	25361	33.8	33.8	33.7	38425	35719	4276
prog2.exe	66672	43.0	43.0	42.9	34724	35653	4916
paper1	29672	48.8	48.8	48.8	38535	33718	6353
paper2	59055	50.2	50.2	50.1	36009	35790	6440
book1	93575	50.7	50.7	50.6	38667	35579	6707
book2	188613	48.9	48.9	48.8	39050	35789	6441
winhlp1.hlp	24099	26.4	26.4	26.3	36513	33942	3570
winhlp2.hlp	83833	14.6	14.6	14.6	36291	32367	2986
word1.doc	68642	49.1	49.1	49.0	32843	35751	6314
word2.doc	200192	58.2	58.2	57.9	36464	36464	7134
AVERAGE	61776	50.4	50.4	50.2	36661	35827	6626

TABLE 4-3 COMPARISON OF RATIOS & SPEEDS

($N = 2^{12}$, $F = 2^{4}$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	60.8	60.8	60.5	33876	33085	4621
readme2	21500	54.5	54.5	54.3	32575	32575	3916
prog1.vb	25546	53.3	53.3	53.0	33613	33176	3419
prog2.vb	48480	53.4	53.4	53.2	31480	31480	3518
prog1.c	58303	70.4	70.4	70.0	34295	33126	6169
prog2.c	29207	67.4	67.4	66.9	33189	33189	5262
prog1.vax	6961	63.1	63.1	62.9	31640	40947	4549
prog2.vax	11453	69.8	69.8	69.4	23373	30139	5614
prog1.exe	25361	34.1	34.1	34.0	32936	35719	2379
prog2.exe	66672	42.5	42.5	42.3	29632	33843	2649
paper1	29672	51.5	51.5	51.4	36185	33718	3699
paper2	59055	53.1	53.1	52.9	31580	33553	3827
book1	93575	53.7	53.7	53.5	34027	32718	3970
book2	188613	52.1	52.1	52.0	35058	33032	3815
winhlp1.hlp	24099	26.1	26.1	26.1	31297	31709	1934
winhlp2.hlp	83833	15.8	15.8	15.7	31754	30595	1655
word1.doc	68642	52.5	52.5	52.4	27131	32843	3775
word2.doc	200192	59.2	59.2	58.8	31427	33702	4032
AVERAGE	61776	51.9	51.9	51.6	31948	33286	3822

TABLE 4-4 COMPARISON OF RATIOS & SPEEDS

($N = 2^{10}$, $F = 2^{5}$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	55.2	55.2	55.2	36876	34039	12896
readme2	21500	44.8	44.8	44.9	39090	35833	10336
prog1.vb	25546	52.3	52.3	52.3	38706	35480	11106
prog2.vb	48480	52.6	52.6	52.6	40066	36727	11300
prog1.c	58303	63.1	63.1	63.1	36439	34295	14949
prog2.c	29207	63.6	63.6	63.6	37931	33189	14751
prog1.vax	6961	59.4	59.4	59.4	43506	40947	13922
prog2.vax	11453	66.9	66.9	66.9	23373	26029	16130
prog1.exe	25361	31.9	31.9	31.9	38425	32936	7708
prog2.exe	66672	41.8	41.8	41.8	35845	35653	8726
paper1	29672	42.4	42.4	42.5	41791	38535	10374
paper2	59055	44.5	44.5	44.5	39634	34738	10659
book1	93575	44.6	44.6	44.7	40684	35445	10842
book2	188613	42.4	42.4	42.4	40388	36552	10501
winhlp1.hlp	24099	24.8	24.8	24.8	39506	33942	6548
winhlp2.hlp	83833	12.2	12.2	12.2	39174	33940	5394
word1.doc	68642	43.7	43.7	43.7	36904	35751	10416
word2.doc	200192	56.7	56.7	56.6	35748	34397	12395
AVERAGE	61776	46.8	46.8	46.8	38005	34913	11053

TABLE 4-5 COMPARISON OF RATIOS & SPEEDS

($N = 2^{11}$, $F = 2^5$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	58.1	58.1	58.1	33876	32182	7964
readme2	21500	50.6	50.6	50.6	32575	35833	6635
prog1.vb	25546	52.5	52.5	52.4	35480	33176	6276
prog2.vb	48480	52.6	52.6	52.5	33902	32756	6438
prog1.c	58303	69.1	69.1	69.1	34295	33315	10411
prog2.c	29207	66.7	66.7	66.6	33571	31405	9331
prog1.vax	6961	61.7	61.7	61.6	31640	31640	7910
prog2.vax	11453	70.2	70.2	70.2	20823	26029	10411
prog1.exe	25361	32.1	32.1	32.2	35223	32936	4313
prog2.exe	66672	42.2	42.2	42.1	31010	34724	4935
paper1	29672	46.2	46.2	46.2	36185	33718	6353
paper2	59055	48.1	48.1	48.0	34535	34738	6518
book1	93575	48.7	48.7	48.7	36269	34151	6707
book2	188613	46.5	46.5	46.5	36910	34355	6465
winhlp1.hlp	24099	24.3	24.3	24.3	36513	33470	3596
winhlp2.hlp	83833	12.5	12.5	12.5	34641	31754	2974
word1.doc	68642	47.2	47.2	47.2	31200	34667	6343
word2.doc	200192	58.3	58.3	58.2	31427	32551	7290
AVERAGE	61776	49.3	49.3	49.3	33338	32967	6715

TABLE 4-6 COMPARISON OF RATIOS & SPEEDS

($N = 2^{12}$, $F = 2^5$)

File Name	File Size (bytes)	Compression Ratio (%)			Compression Speed (c/s)		
		LZSS	LZAVL	LZKMP	LZSS	LZAVL	LZKMP
readme1	70802	62.1	62.1	62.0	16128	16738	4145
readme2	21500	55.2	55.2	55.2	17768	18695	3434
prog1.vb	25546	54.3	54.3	54.3	17864	17864	3019
prog2.vb	48480	54.6	54.6	54.6	18022	18022	3119
prog1.c	58303	72.8	72.8	72.7	16331	16331	5677
prog2.c	29207	69.3	69.3	69.3	15211	16136	4788
prog1.vax	6961	64.4	64.4	64.3	12656	17848	4070
prog2.vax	11453	71.5	71.5	71.5	8359	13967	5205
prog1.exe	25361	34.2	34.2	34.2	18377	19212	1990
prog2.exe	66672	42.9	42.9	42.9	15152	17826	2356
paper1	29672	50.9	50.9	50.9	20749	19267	3275
paper2	59055	52.8	52.8	52.8	17628	18570	3382
book1	93575	53.8	53.8	53.8	18752	18134	3491
book2	188613	51.8	51.8	51.8	19187	18276	3383
winhlp1.hlp	24099	25.3	25.3	25.3		19916	1693
winhlp2.hlp	83833	14.8	14.8	14.8	20347	19052	1404
word1.doc	68642	52.7	52.7	52.7	14889	18402	3333
word2.doc	200192	60.7	60.7	60.7	14289	16118	3738
AVERAGE	61776	52.4	52.4	52.4	16508	17799	3417

To analyze the performance difference of the LZSS and the LZAVL, we also measured and compared the "average search length" in those two tree structures. This value indicates how many nodes, on the average, the scheme needs to search for an insertion (deletion is the same.) Theoretically, the better the tree is height balanced, the less the average search length should be, regardless what kind of text file is being compressed, and the "average search length" should be almost a constant because the dictionary tree is a fixed size.

The results from the above experiments are compared in Tables 4-7 and 4-8 as follows:

TABLE 4-7 COMPARISON OF SEARCH LENGTH

File Name	File Size (bytes)	(N=10, F=4)		(N=11, F=4)		(N=12, F=4)	
		LZSS	LZAVL	LZSS	LZAVL	LZSS	LZAVL
readme1	70802	11	9	13	10	14	11
readme2	21500	13	9	15	10	17	11
prog1.vb	25546	13	10	14	11	16	12
prog2.vb	48480	12	10	14	11	16	12
prog1.c	58303	11	9	12	10	14	11
prog2.c	29207	11	9	12	10	14	11
prog1.vax	6961	14	9	16	10	21	11
prog2.vax	11453	19	9	22	10	25	11
prog1.exe	25361	12	10	15	11	17	12
prog2.exe	66672	14	9	17	10	21	11
paper1	29672	12	10	13	11	15	12
paper2	59055	12	10	15	11	19	12
book1	93575	11	9	13	11	14	11
book2	188613	12	10	13	11	14	12
winhlp1.hlp	24099	12	10	14	11	16	12
winhlp2.hlp	83833	12	10	13	11	15	12
word1.doc	68642	14	10	20	11	26	12
word2.doc	200192	12	9	14	10	18	11
AVERAGE	61776	12.6	9.5	14.7	10.6	17.3	11.5

TABLE 4-8 COMPARISON OF SEARCH LENGTH

File Name	File Size (bytes)	(N=10, F=5)		(N=11, F=5)		(N=12, F=5)	
		LZSS	LZAVL	LZSS	LZAVL	LZSS	LZAVL
readme1	70802	12	10	14	11	16	12
readme2	21500	13	10	16	10	18	11
prog1.vb	25546	13	10	14	11	16	12
prog2.vb	48480	12	10	15	11	16	12
prog1.c	58303	12	9	13	10	15	11
prog2.c	29207	12	10	13	11	15	11
prog1.vax	6961	15	9	19	10	27	11
prog2.vax	11453	22	10	28	10	34	11
prog1.exe	25361	12	10	15	11	17	12
prog2.exe	66672	15	9	18	10	24	11
paper1	29672	12	10	13	11	15	12
paper2	59055	12	10	15	11	19	12
book1	93575	12	9	13	11	15	12
book2	188613	12	10	13	11	14	12
winhlp1.hlp	24099	12	10	14	11	16	12
winhlp2.hlp	83833	12	10	13	11	15	12
word1.doc	68642	14	10	21	11	26	12
word2.doc	200192	13	9	15	11	19	12
AVERAGE	61776	13.2	9.7	15.7	10.7	18.7	11.7

4.4 EXPERIMENT RESULTS ANALYSIS

Compression Ratio: These values increase as the text window size increases, but they do not necessarily increase as the look-ahead buffer size increases. The possible reasons are:

(1) When the text window (dictionary) size increases (the look-ahead buffer size remains the same), the probability of finding a longer match increases, and the compression rate thus increases. Even though we have to use more bits to represent the match position, which will decrease the compression, we still gain more than we lose.

(2) When the look-ahead buffer size increases, even longer matches can be found to increase the compression. But if the dictionary size is small, the probability of finding

longer matches is still small. In those cases, we don't make full use of the larger look-ahead buffer, and we gain less than we lose. In the next Chapter, we will discuss this problem further.

So, if both the dictionary size and the look-ahead buffer size are carefully selected to increase, we can expect to gain a better compression.

Compression Speed: In the following chart (Figure 4-8), we give an overall speed comparison of different schemes with different dictionary size and look-ahead buffer size.

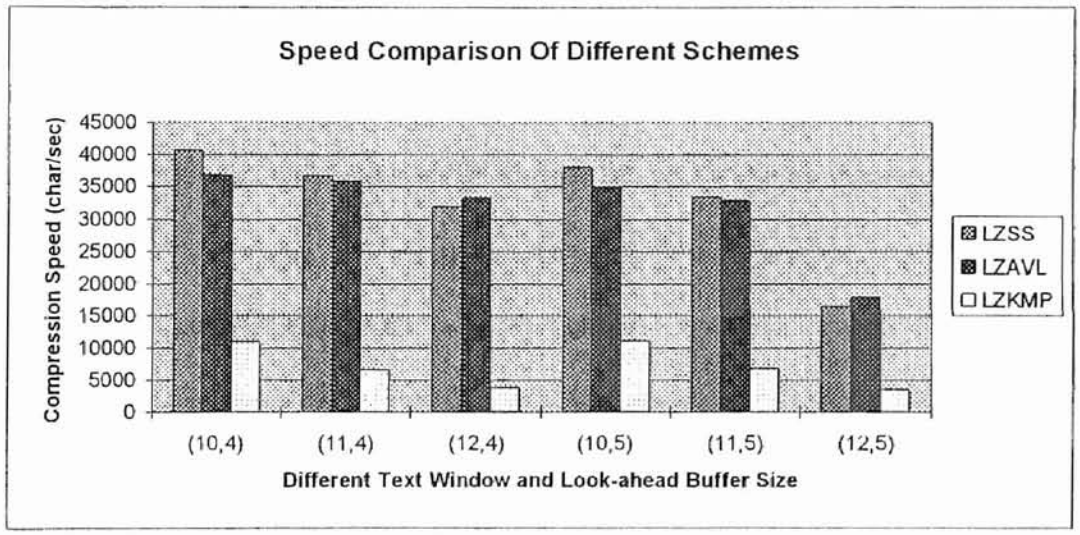


Figure 4-8 Compression Speed Comparison of Different Schemes

Among them, the LZKMP has the poorest performance. This is quite reasonable because its time complexity in finding the longest match is $O(n+m)$, a linear relation with the dictionary size n. When n becomes much larger, the speed of this scheme will become very slow. The only advantage of this scheme is that it uses much smaller amount of

memory than that of the LZSS or the LZAVL. It can be applied to a special case where memory is very limited and compression speed is not important.

The original LZSS variant outperforms the LZAVL when the text window size is small (1K and 2K). But when the dictionary gets larger and larger (e.g., 4K), the LZAVL outperforms the original LZSS. This is our expected result, because in the world of data compression, a better scheme means it has a better compression and a higher speed as well. Memory usage is another consideration, but for LZ77 variants, which use only a fixed size text window as the dictionary, the memory requirement is relatively low.

Now we will explain how and why the LZAVL can outperform the LZSS.

The original LZSS scheme uses a binary search tree to store all the phrases (the tree size is equal to the text window size). During the compression, insertion and deletion are very frequently necessary because data stream is sliding in and out very frequently, so the time used for searching the dictionary tree will be a bottleneck factor that affects the speed. If the binary tree is height unbalanced, the overall time used for tree searching is most likely more than that used for a height balanced tree because so many "worst cases" are met. To judge the balance condition of a tree, we use the "average search length" for a heuristic measure.

Tables 4-7 and 4-8 show the average search length for different dictionary size and look-ahead buffer size of the LZSS and the LZAVL algorithms. We notice that the average search lengths for the two schemes obey the following rules:

(1) For the same N and F, the LZSS always has a longer average search length, and the difference becomes larger as N becomes larger. This means the difference in the

time used for tree searching by LZSS and LZAVL becomes larger as the dictionary tree is getting larger. The average tree searching time for LZAVL does not change greatly while it changes greatly for LZSS.

(2) The average search length for LZAVL almost remains the same for a given dictionary and a given look-ahead buffer, no matter what type of file is being compressed. But for LZSS, this value changes a lot for different input files. For example, when N=2^12, F=2^5, for LZAVL, the average search length is 11 or 12 for any input file; but for LZSS, the file "book2" has a shortest average search length of only 14 while the file "prog2.vax" has a largest value of 34!

This can be explained by the nature of the AVL tree. LZAVL uses an AVL tree to maintain the dictionary, no matter what kind of file it is compressing, and the tree is always "height balanced" so the average search length is almost the same, as long as the dictionary size is fixed. LZSS, on the other hand, uses an ordinary binary search tree to maintain the dictionary, which in most cases is unbalanced (the balance condition depends on the file.) When the dictionary is large, this unbalance is more serious.

So, why in all those smaller dictionary cases, does LZSS compress faster than LZAVL? The possible answer is: in a relatively small dictionary tree, no matter whether it is balanced or not, the time used for tree searching is short. The difference in search time for the two schemes is less than the time used by LZAVL for maintaining the tree height balance after node insertions and deletions. This causes the original LZSS to run faster than LZAVL.

If the tree is sufficiently large, things are different. The time saved from tree searching becomes longer than that spent on maintaining the tree balance, so the overall speed of the LZAVL finally outperforms that of the original LZSS.

Take the results in Tables 4-3 and 4-6 for example. Among the 18 test files, for N=2^12 and F=2^4, the compression speed of LZAVL for all total 12 files is greater than or equal to that of LZSS; for N=2^12 and F=2^5, things are even better, the number is 14.

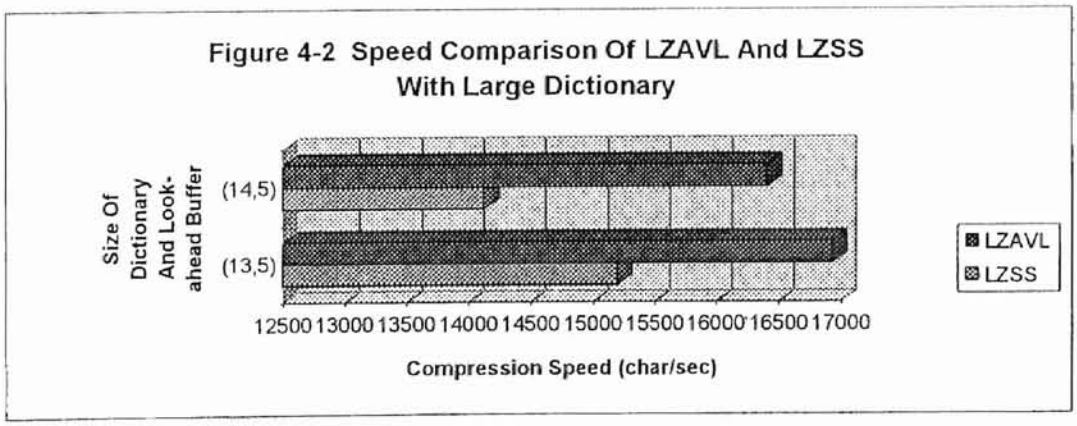
Fortunately, some of the selected test files can represent the "worst cases". "prog2.vax" is such a file because it has the longest average search length. This "worst case" file can have a better compressing speed by using LZAVL.

We should point out here that the binary tree used by LZSS is a dynamically changed binary tree. The data stream to be compressed slides in and out on the fly such that when the tree is small, the overall "shape" of that tree is closed to height balanced. This prediction can be confirmed by comparing the average search lengths of LZSS and LZAVL for N=2^10 (this value only changed slightly for different input files). But when the tree is very large, the overall shape of the height unbalanced tree will remain for a relatively longer time, which eventually decreases the compression speed.

We further tested two cases with even larger dictionary sizes (8K and 16K) and list the effects in Table 4-9. Comparing the compression ratios with the previous ones, we see that when the dictionary size increased from 8K to 16K, the compression ratio only increased a little bit (+0.7%) but from 4K to 8K the increment was larger (+1.7%), and that is why the dictionary size of LZ77 variations is usually selected as 8K (or 4K).

TABLE 4-9 COMPARISON OF RATIOS & SPEEDS

File Name	File Size (bytes)	C. R (%) (N=13,F=5)		C. S (char/sec) (N=13,F=5)		C. R. (%) (N=14,F=5)		C. S (char/sec) (N=14,F=5)	
		LZSS	LZAVL	LZSS	LZAVL	LZSS	LZAVL	LZSS	LZAVL
readme1	70802	63.9	63.9	14332	15733	65.7	65.7	12757	15000
readme2	21500	57.1	57.1	16287	17916	57.7	57.7	16287	17768
prog1.vb	25546	54.3	54.3	16066	17864	53.5	53.5	14939	17260
prog2.vb	48480	54.9	54.9	16323	16951	55.3	55.3	14690	16323
prog1.c	58303	76.3	76.3	14325	15143	78.2	78.2	12927	14360
prog2.c	29207	71.8	71.8	12643	15133	71.9	71.9	11824	14387
prog1.vax	6961	63.3	63.3	11601	18318	62.1	62.1	11601	18318
prog2.vax	11453	71.5	71.5	7437	14874	70.7	70.7	7203	14874
prog1.exe	25361	35.1	35.1	17135	18511	34.7	34.7	16468	19359
prog2.exe	66672	42.8	42.8	17591	15988	42.3	42.3	16421	15187
paper1	29672	53.2	53.2	18661	18092	54.4	54.4	17454	17983
paper2	59055	55.6	55.6	15378	17628	57.2	57.2	13607	16542
book1	93575	59.8	59.8	16532	16709	61.2	61.2	14948	15621
book2	188613	56.6	56.6	16915	16840	59.3	59.3	15334	15823
winhlp1.hlp	24099	25.0	25.0	18256	19126	25.0	25.0	18396	19126
winhlp2.hlp	83833	15.6	15.6	18145	17761	16.1	16.1	16215	17143
word1.doc	68642	56.1	56.1	13149	17117	58.1	58.1	11459	16265
word2.doc	200192	61.4	61.4	12783	14939	62.1	62.1	11639	13960
AVERAGE	61776	54.1	54.1	15198	16925	54.8	54.8	14121	16406



4.4 MEMORY USAGE FOR EACH SCHEME

Apparently, among the three schemes, LZKMP uses the smallest amount of memory during the compression because it does not need a sophisticated data structure to maintain the dictionary tree, but the tradeoff is that its speed is too slow.

In our programming of the LZAVL variant, we use an extra component, the balance scale code, in each node in the tree, which consumes a bit more memory than that of the original LZSS. But actually, we can make it use the same memory as the LZSS for the following reasons:

(1) By comparing the performances of a given scheme with different dictionary sizes, we know that the appropriate size for the dictionary is no more than 16K; thus all the pointers in a node of the tree never contain a number larger than 16K (2^14). In other words, 14 bits is already enough to represent the pointers.

(2) Because of the size of the dictionary, all pointers in a node need to use an integer to keep the index information. An integer, no matter what type it is, usually is at least 16 bits in length. We can make full use of the other two bits by embedding the balance scale code (it happens to need only two bits) into a pointer in each node, for example, the parent pointer.

CHAPTER V

CODING REDUNDANCY

As we already knew, LZ77, LZ78, LZSS and LZW all used fixed-size pointers regardless of the length of the phrase they represent. For instance, a pointer to position 40 and a pointer to position 4000 will each use 12 bits to represent the indexes. This is obviously wasting output space. Moreover , in practice some phrase lengths are much more likely to occur than others, and better compression can be achieved by reducing the redundancy in the output codes themselves. This can be achieved by allowing variable sized pointers by using multiple flags.

LZB [Bell87] is one of the best of the LZ77 compressors, and it uses a variable length encoding method to achieve a compact representation. But that scheme is somewhat complicated. In this chapter, we are going to try a simple scheme for compacting the pointer representation.

5.1 A TWO-BIT FLAG SCHEME FOR LZSS

Each encoding of the original LZSS starts with a one-bit flag which tells the decoder its succeeding content is a literal or a phrase. By adding one more flag bit, we can specify more detailed information about the length of the pointers.

Suppose $k = \text{Log}(N)$, where N is the text window size (we usually select a power of 2 as N for a full use of the pointer length.) There are two choices to use for the these two flags shown below:

Flag 1	Flag 2	Meaning
0	0	succeeding a character
0	1	succeeding a pointer tuple in which the "offset" length is k bits
1	0	succeeding a pointer tuple in which the "offset" length is k' bits
1	1	succeeding a pointer tuple in which the "offset" length is k'' bits

Flag 1	Flag 2	Meaning
0		succeeding a character
1	0	succeeding a pointer tuple in which the "offset" length is k bits
1	1	succeeding a pointer tuple in which the "offset" length is k' bits

In the above schemes, the total number of bits reserved for the match length will not be changed because it is relatively a small number and contains few "redundant" bits.

Choice one fully uses two bits for the flag. But we will have to use 10 bits (more than 10 percent of the original 8 bits) to encode a single character; choice two is expected to gain a better result than choice one. In the work of this chapter, we only try the second scheme.

The scheme selected is based on the following investigation:

In LZSS, the occurrence of the longest match that can be found in the text window is not necessarily the only one. From the view of the decoder, it doesn't care which position is to be picked up because the decoding process is only a copy process from the previous decoded text as long as it can find the content needed. But a possible optimal way to pick up the position in the above two schemes is that we choose the one with the

smaller index, by which means the index may be represented using a smaller number of bits.

5.2 EXPERIMENTAL RESULTS

The results for one-bit flag LZSS and the two-bit flag scheme are compared in Tables 5-1 and 5-2.

In the two tables, N (or N') and F are numbers of the power of 2 of the text window size and look-ahead buffer. i.e. window size = 2^N and buffer size = 2^F . N' is the "break-point" for the second bit flag. e.g. if offset < $2^{N'}$, set the second flag to 1; otherwise, set it to 0 for $2^{N'} \leq$ offset < 2^N.

TABLE 5-1 COMPRESSION RATIOS OF ONE-BIT FLAG AND TWO-BIT FLAG SCHEMES

File Name	File Size (bytes)	(N=12,F=4)	(N'=8)	(N'=9)	(N'=10)
		LZSS	LZSS2	LZSS2	LZSS2
readme1	70802	60.8	59.7	60.1	60.3
readme2	21500	54.5	53.7	54.3	54.4
prog1.vb	25546	53.3	52.1	52.5	52.7
prog2.vb	48480	53.4	52.3	52.7	52.8
prog1.c	58303	70.4	69.6	69.8	70.0
prog2.c	29207	67.4	66.6	66.8	66.9
prog1.vax	6961	63.1	62.9	63.0	63.2
prog2.vax	11453	69.8	69.2	69.4	69.5
prog1.exe	25361	34.1	32.5	32.9	33.2
prog2.exe	66672	42.5	41.4	41.8	42.0
paper1	29672	51.5	50.7	51.2	51.3
paper2	59055	53.1	51.8	52.3	52.6
book1	93575	53.7	52.4	52.9	53.2
book2	188613	52.1	50.7	51.2	51.6
winhlp1.hlp	24099	26.1	24.9	25.4	25.6
winhlp2.hlp	83833	15.8	14.4	14.7	15.0
word1.doc	68642	52.5	51.4	51.8	52.0
word2.doc	200192	59.2	58.1	58.4	58.6
AVERAGE	61776	51.9	50.8	51.2	51.4

TABLE 5-2 COMPRESSION RATIOS OF ONE-BIT FLAG AND TWO-BIT FLAG SCHEMES

File Name	File Size (bytes)	(N=12,F=5) LZSS	(N'=8) LZSS2	(N'=9) LZSS2	(N'=10) LZSS2
readme1	70802	62.1	61.2	61.4	61.6
readme2	21500	55.2	54.5	54.8	54.9
prog1.vb	25546	54.3	53.4	53.6	53.8
prog2.vb	48480	54.6	53.8	54.1	54.1
prog1.c	58303	72.8	72.2	72.3	72.4
prog2.c	29207	69.3	68.7	68.9	69.0
prog1.vax	6961	64.4	63.8	64.0	64.3
prog2.vax	11453	71.5	71.0	71.2	71.2
prog1.exe	25361	34.2	33.3	33.4	33.6
prog2.exe	66672	42.9	42.2	42.4	42.6
paper1	29672	50.9	50.0	50.4	50.5
paper2	59055	52.8	51.6	51.9	52.2
book1	93575	53.8	52.7	53.0	53.2
book2	188613	51.8	50.6	51.0	51.3
winhlp1.hlp	24099	25.3	24.5	24.8	24.9
winhlp2.hlp	83833	14.8	14.2	14.3	14.5
word1.doc	68642	52.7	51.7	51.9	52.2
word2.doc	200192	60.7	59.9	60.1	60.2
AVERAGE	61776	52.4	51.6	51.9	52.0

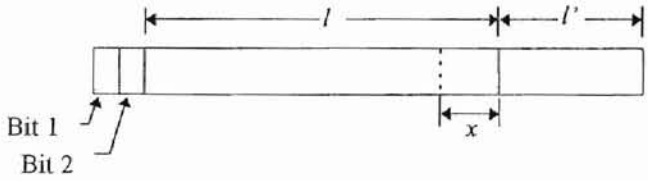
5.3 PERFORMANCE ANALYSIS

The experimental results show that the two-bit flag scheme did not successfully improve on the original one-bit scheme. In its best case, it only outperformed the other scheme by a very little bit. e.g., "prog1.vax" with N'=10.

In the following, we try to explain why this new scheme does not work better than the old scheme based on probability theory and two reasonable assumptions.

Assumption 1: The longest match between the text window and the look-ahead buffer can occur at any position in the text window with equal probability (except inside the look-ahead buffer, where it cannot occur).

Assumption 2: The longest match can occur more than once in the text window. The maximum match length is limited by the buffer size. The smaller the buffer, the larger the probability of more longest matches.



Bit1: The original bit as a flag in LZSS
Bit2: The addition bit as a second flag in the two-bit flag scheme. According to its value, the total number of bits used to represent the offset can be *l* or *l-x*. The total number of bits used for the match length *l'* won't change in both schemes.

Figure 5-1 Calculate the possible improvment of the two-bit scheme.

We assume the full length of the original "offset" pointer is *l*. Using the above two-bit scheme, the "break-point" is set at position *l-x*. Thus the probability ratio of using pointers of length *l-x* bits over that of using pointers of length with *l* bits is $2^{(l-x)}/2^{l} = 1/2^{x}$.

In our experiment, we use $l = 12$, $l-x = 8, 9, 10$; so $x = 4, 3, 2$.

If $x = 3$, the probability ratio is 1/8. This means that if the longest match occurs in the text window at eight places, we can at least pick up the one with the smallest offset to represent it in *l-x* bits.

Because this two-bit flag scheme needs one extra bit for every phrase output code, in each successful applying of this two-bit scheme for one output, we can save 2 bits (x-1), otherwise, we will waste one more bit. If the longest match occurs eight times for each

parsing, we will absolutely save output space. If it occurs four times, we still absolutely save output space since the probabilities of saving 2 bits and wasting 1 bit are both 50%. By further calculation, we know that for $x = 3$, only if the longest match occurs more than 2.7 times in each parsing can we save output space.

Similarly, we calculate the probability and find if $x = 4$, we need an average of four occurrences in each parsing to guarantee the output saving. $x = 2$ is the best case which needs the smallest number of longest match occurrences for this two-bit flag scheme, but it still needs at least 1.25 times to guarantee the compression. This means that for every four parsings, at least one of them has to find two longest matches.

As the matter stands, the experimental results have shown that on the average the probability of finding more than one longest match in each parsing was very low. That is why the compression increased when the x value decreased. If we decrease the size of the look-ahead buffer (the maximum length of the longest match is limited to this size), we can expect to find more occurrences but at the same time it may also degrade the compression. The experiment showed that the smaller look-ahead buffer only gave worse performance.

CHAPTER VI

PARSING STRATEGIES

As mentioned before, once a dictionary has been chosen, there is more than one way to choose which phrases in the input text will be replaced by indexes to the dictionary. The task of splitting the text into phrases for coding is called *parsing*. We can simply classify parsing strategies into two: *greedy* and *non-greedy*. "Greedy" algorithms don't look ahead into the input stream to analyze it for the best combination of indexes and characters, while "non-greedy" algorithms need to look ahead into the input stream to do this kind of analysis.

The following diagram is used to illustrate these ideas:

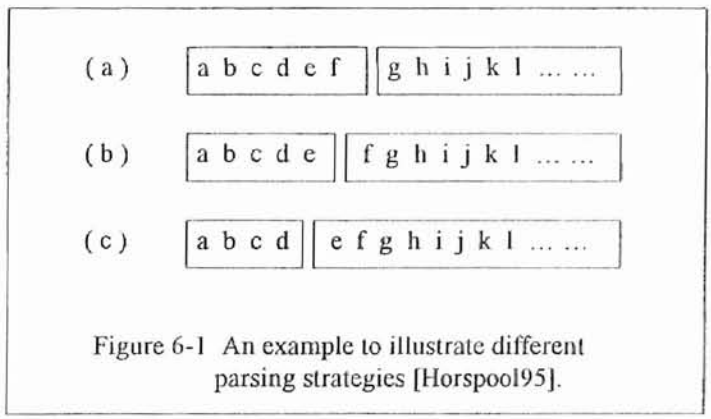


Figure 6-1 An example to illustrate different parsing strategies [Horspool95].

Case a: Assume that the longest match between look-ahead buffer and text window is "abcdef". If the "greedy" parsing strategy is used, the encoder will output a pointer tuple and then slide the window to position 'g'. If the next longest match is "gh", it will output another pointer tuple for that match. The total length in two passes will be "abcdefgh".

Case b: If the encoder uses a "non-greedy" parsing strategy to determine the break-point, it finds that if the current output is "abcde" instead of "abcdef", the next longest match can be (much) longer, for instance, "fghijk". In this case we can say, by looking ahead one pass, the output becomes somewhat "optimized" because "abcde" + "fghijk" > "abcdef" + "gh".

Case c: If the encoder can manage to look ahead into the entire coming input stream, and it can determine each "break-point" so that the average match length is the longest, and it can thus guarantee to generate the best compression. But obviously, this scheme is unpracticed.

6.1 A "NON-GREEDY" PARSING SCHEME

For the LZ class of methods, a greedy parsing approach is commonly used because it is an easily implemented approach that achieves excellent results [Bell94]. Some authors even claimed that it is the only realistic approach for practical text compression applications [Gonzalez85] [Schuegraf74].

In the following, we are going to analyze a newly presented "Non-Greedy" parsing scheme [Horspool95]. The author modified the original LZW and LZSS variants using a "Non-Greedy" parsing strategy. In this chapter, we are not going to implement the program. Instead, we do some theoretical analysis on it.

The paper gave both its compression and decompression algorithms. We mainly discuss its compression because the decompression procedure for LZ schemes is always fast. The compression algorithm is listed in Figure 6-2.

As the original paper said, in the above figure, α++b represents the new string formed by appending one character b to the string α, the length of α is obtained by the function length(α), the function head(α) returns the first symbol of string α, and the function prefix(α,i) returns a substring composed of the first i symbols of the string α. K is a parameter that limits the number of non-greedy parsing possibilities considered at each step.

Line 1 does dictionary initialization. For the LZW algorithm, the dictionary is pre-loaded with 256 characters so that any ASCII symbol in the input file can find a match in the dictionary.

```
1.  Initialize the dictionary D with all strings of length 1;
2.  set α =  the string in D that matches the first symbol of the input;
3.  set L = length(α);
4.  while more than L symbols of input remain do
5.  begin
6.     for j := 0 to max* (L-1, K) do
7        find β_j, the longest string in D that matches the input starting L-j
           symbols ahead;
8.     add the new string α++head(β_0) to D;
9.     set j = value of j in range 0 to max*(L-1,K) such that
         L-j + length(β_j) is maximum;
10.    output the index in D of the string prefix(α,j);
11.    advance j symbols through the input;
12.    set α = β_j;
13.    set L = length(α);
14. end;
15. output the index in D of string α;
```

Figure 6-2 The Non-Greedy LZW Compression Algorithm [Horspool95].
*: this should be "min" instead of "max"

Line 2 finds the first longest match α in the dictionary. This is a start-up step.

Line 3 indicates that this algorithm employs a variable length of buffer to keep the next longest match string, the length of which is the length of α, L. Whenever there exist L symbols in the buffer, the loop continues.

Line 5 tries to find the "break-point" for the next longest match β_j where j ranges from 0 to min(L-1, K). But this step takes too much time. It needs at least min(L-1, K)+1 times of searching the dictionary to find βj, the next longest match string. The algorithm uses a parameter K to avoid an unbounded number of operations in this searching because when the dictionary gets larger and larger, the probability of finding a longer match (the length of L) becomes larger and larger.

Lines 6 and 7 add the longest match plus the first new symbol into the dictionary as the original LZW algorithm does (in a greedy way).

Line 8 is actually in the loop of line 6 and 7.

Line 9 outputs the left part of the break-point.

Lines 10,11 reset α to be β_j and calculate the length of the new α.

Line 12 continues the loop beginning from line 4

Line 13 outputs the last string.

This "Non-Greedy" parsing algorithm is actually inefficient based on the following two reasons:

(1) For each parsing, it needs to look up the dictionary min(L-1, K) times. As L can become larger and larger, the look-up times will be limited to K but this is an artificially set parameter, which has no theory support.

(2) To avoid an unbounded time required for the parsing, the algorithm employs a variable length buffer and only look ahead by one parsed string when choosing its course of action. But this seems somewhat unreasonable. Because the next longest match β_j , even though waits much effort to be allocated, still cannot guarantee its achievement. In the third pass, β_j becomes α and this will be shortened by its second optimal process. Intuitively, the smaller the look ahead buffer used, the smaller the number of break-points we have to decide, the larger the probability we will lose the optimizing effort.

This algorithm can be viewed as a two-pass greedy parsing scheme. But unlike the original greedy scheme, it cannot guarantee its effort. That is a big problem of this "Non-Greedy" parsing algorithm.

6.2 TIME COMPLEXITY ANALYSIS

The main inefficient step of this "Non-greedy" algorithm lies in line 6 and 7. The author must have noticed this loop wastes too much parsing time so he has used a parameter K to limit the "optimal possibilities", even for a two pass optimization. Otherwise, if the program tried to find the best "break-point" each step in α, as the dictionary getting larger and larger, L (the longest match) can be a very large number, and the parsing process will be slower and slower. In the worst case, each time it needs to do K searches in the dictionary, and each dictionary search needs $K+L$ searching length. The upper bound for each step will be $O(K*(K+L))$. As L increases, this can be a very large number.

6.3 COMPARISON AND CONCLUSION

If compared with a greedy parsing strategy, this "Non-greedy" parsing scheme has the following potential problems:

First, it would be very slow, especially when the file is more "compressible". That is, when the longest match L becomes larger, the time used for the "optimizing step" may be quite long.

Second, even though the "look-ahead buffer" size could be as large as the length of α, this size is still too small. The optimal effort cannot be guaranteed for the next step.

CHAPTER VII

SUMMARY, CONCLUSIONS, AND SUGGESTIONS FOR FUTURE WORK

7.1 SUMMARY AND CONCLUSIONS

The efficiency problems for LZ adaptive dictionary compression schemes can be basically classified into three categories: The efficiency of finding the longest match between the dictionary and the to be encoded data stream; the redundancy of coding; and the optimal parsing for the output. This thesis has focused on these three problems by trying new ways to improve the performance and analyzing an existing approach.

The Knuth-Morris-Pratt algorithm can be modified for finding the longest match. This algorithm can be applied in a data compression program in the case that modest memory is available but the compression speed is not important. If the ratio is also not very important, this scheme can also gain an acceptable compression speed. But the overall performance is worse than LZSS and LZAVL, as predicted.

LZSS uses a special data structure, a binary search tree, to maintain the dictionary, and gains significant improvement in performance in finding the longest match. Its speed is much faster than that of the LZKMP, especially in the case where we increase the text window size to increase the compression ratio.

LZAVL employs a new approach, the AVL tree, to maintain the dictionary of LZSS. As the experiment results show, when the dictionary tree is not so large, LZSS outperforms LZAVL, but whenever a larger dictionary tree size is used to achieve better

compression, LZAVL outperforms LZSS. In LZAVL, even though extra time is spent to maintain the tree balance, it still improves the overall performance by avoiding the worst cases of searching.

The two-bit flag schemes for LZSS do not outperform the original one-bit flag schemes. The main reason is a probability problem. There exists a contradiction in finding a longer match and expecting it to occur at more than one place as well.

"Greedy" Parsing is still a good parsing strategy in practice. Any complete "Optimal" parsing wastes too much time while the newly presented sub-optimal scheme, namely "Non-greedy" parsing, cannot guarantee its performance and is still low speed. The effort is not worthwhile.

7.2 SUGGESTIONS FOR FUTURE WORK

We suggest that the following three points can be included in further research:

(1) There may exist some BM algorithm variant that can be modified and applied to finding the longest match and yet has an attractive running time of $O(n/m)$. If such a variant can be found, we should use it in a data compression program and an expected good performance may be achieved.

(2) We need to further optimize and simplify the rebalancing for insertion and deletion routines of the AVL version and try to minimize the memory usage by embedding the "balance scale" in another pointer in the node structure of an AVL tree. The expected result is that LZAVL can replace the original LZSS in any cases.

(3) For the LZW variant, the encoder and decoder build their own dictionary. The two dictionaries are almost the same except in some special cases (the KωKωK problem,

for instance.) during compression and decompression. If we manage to find a way to guarantee that these two dictionaries are exactly the same, we will be able to reduce the output file size by limiting each output code length to log(d) bits (d is the current dictionary size).

SELECTED BIBLIOGRAPHY

[Bell94] Bell T.C, and Witten, I.H. "The Relationship between Greedy Parsing and Symbolwise Text Compression. Journal of ACM 41, 4 (July 1994), pp. 708-724.

[Bell90] Bell T.C., Cleary J.G., Witten I.H., "Text Compression", Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Bell86] Bell T.C., "Better OPM/L Text Compression", IEEE Transactions on Communications, VOL. COM-34, No. 12, December 1996, pp.1176-1182.

[Boyer77] Boyer R.S., Moore J.S., "A Fast String Searching Algorithm", *Communications of the ACM*, Vol. 20, No. 10, 1977, pp.762-772.

[Brent87] Brent R.P., "A Linear Algorithm for Data Compression", *The Australian Computer Journal*, Vol. 19, No. 2, May 1987, pp.64-68.

[Corman90] Corman T.H., "Introduction to Algorithms", McGraw-Hill, NewYork, 869-883, 1990, pp.869-883.

[Debra87] Debra A. Lelewer and Daniel S. Hirschberg, "Data Compression", *ACM Computing Surveys*, Vol. 19, No. 3, September 1987, pp.261-269.

[Fiala89] Fiala E.R., Greene D.H., "Data Compression with Finite Windows", *Communications of the ACM*, Vol. 32, 1989, pp.490-503.

[Gonzalez85] Gonzalez-Smith M.E., and Storer, J.A. "Parallel Algorithms for Data Compression." Journal of ACM 32, 2 (April 1985), pp. 344-373.

[Knuth77] Knuth D.E., Morris J.H., Jr and Pratt V.B., "Fast pattern matching in strings", *SIAM J. Computing*, 6, 1977, pp.323-350.

[Horspool95] Horspool R.N., "The Effect of Non-Greedy Parsing in Ziv-Lempel Compression Methods", *Data Compression Conference*, 1995, pp.303-311.

[Hume91] Hume A., "Fast String Searching", *Software-Practice and Experience*, Vol.21(11), 1221-1248 (November 1991).

[Jakobsson85] Jakobsson M., "Compression of Character Strings by An Adaptive Dictionary", *BIT* 25(1985), pp.593-603.

[Nelson96] Nelson M., "Data Compression Book", 2nd Edition, Jean-Loup Gailly, 1996.

[Rodeh81] Rodeh M., Pratt V.R., and Even S., "Linear Algorithm for Data Compression Via String Matching", *J. ACM* 28,1 (Jan, 1981), pp.16-24.

[Reingold86] Reingold E. M., Hansen W. J., "Data Structure", Little, Brown and Company, 1983, pp.302-313.

[Schegraf74] Schuegraf E.J., and Heaps, H.S. "A Comparison of Algorithms for Database Compression by use of Fragments as Language Elements." Inf. Stor. Ret. 10(1974), pp.309-319.

[Semba85] Semba I., "An Efficient String Searching Algorithm", *Journal of Information Processing*, Vol. 8, No. 2, 1985, pp.101-109.

[Smith91] Smith P.D., "Experiments with a Very Fast Substring Search Algorithm", *Software-Practice and Experience*, Vol. 21(10), 1065-1074 (October, 1991).

[Storer88] Storer J.A., "Data Compression: Methods and Theory", Computer Science Press, 1803 Research Boulevard, Rockville, Maryland 20850, 1988.

[Welch84] Welch, T.A., " A Technique for High-Performance Data Compression", *Computer* 17, 6 (June, 1984), pp.8-19.

[Williams90] Williams R.N., "Adaptive Data Compression", Kluwer Academic Publishers, 1990, pp.8-10.

[Williams91] Williams R.N., "An Extremely Fast ZIV-Lempel Data Compression Algorithm", *Data Compression Conference*, 1991, pp.363-371.

[Zhu87] Zhu R.F., Takaoka T., "On Improving the Average Case of the Boyer-Moore String Matching Algorithm", Journal of Information Processing, Vol. 10, No. 3, 1987, pp.173-177.

[Ziv77] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp.337-343.

[Ziv78] Ziv J., Lempel A., "Compression of Individual Sequences via Variable Rate Coding", *IEEE Transactions on Information Theory*, Vol.24, No. 5, pp.530-536.

VITA

LIN KE

Candidate for the Degree of

Master of Science

Thesis: A STUDY OF EFFICIENT PARSING IN LZ ADAPTIVE DICTIONARY COMPRESSION

Major Field: Computer Science

Biographical:

Personal Data: Born in Zhanjiang, Guangdong, P.R. China, January 12, 1965, the son of Mr. Zhenwei Ke and Mrs. Qiongzhen Lin

Education: Graduated from Zhanjiang No.2 High School, Zhanjiang, Guangdong, P.R.China, in July, 1982; received Bachelor of Science degree in Chemistry from Zhongshan University, Guangzhou, Guangdong, P.R. China in July, 1986; received Master of Science degree in Physical Chemistry from Zhongshan University, Guangzhou, Guangdong, P.R. China in July, 1992; completed requirements for Master of Science at Oklahoma State University in May, 1997.

Professional Experience: Information Engineer, Information Center of Petrochemical Industry Bureau, Guangzhou, Guangdong, P.R. China, 1992-1994; Computer Professor Assistant, Guangdong Medical College, Zhanjian, Guangdong, P.R. China, 1986-1989.