Order Number 9325077

String pattern matching and lossless data compression

Chang, Daniel Kuo-Yee, Ph.D.

City University of New York, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

String Pattern Matching

and

Lossless Data Compression

by

Daniel K. Chang

A dissertation submitted to the Graduate Faculty in
Computer Science in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy,
The City University of New York

1993

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Apr. 12 '93
Date

Chair of Examining Committee

April 28 '93
Date

Executive Officer

Prof. Michael Anshel

Prof. Stanley Habib

Dr. Victor Miller
Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

String Pattern Matching

and

Lossless Data Compression

by

Daniel K. Chang

Advisor: Professor Charles Giardina

A new string pattern matching algorithm has been designed that is better than the Boyer-Moore algorithm for certain types of patterns. It is particularly very efficient to search a non case sensitive pattern or pattern with don't care symbols because the extra works are all done in preprocessing the pattern on-the-fly, the search part of the algorithm doesn't need to do any conversion or extra work. String pattern matching is used for many applications, including text editing, information retrieval and lossless data compression.

A new lossless data compression/expansion algorithm has been designed that overcomes several shortcomings of the best algorithms known. The algorithm uses multiple dictionaries for storing previously encountered strings of a data file to be compressed. Each of these dictionaries (one is a short one and needs less bits for indexing an entry) is initialized with frequently occurring strings. An input data file is compared with the dictionaries so that the longest substring which can be found in a dictionary will give its position in the dictionary as an output code. Compression and

expansion use the same procedure to form and update dictionaries. A compressed result consists of a series of pointers to the dictionaries. During compression/expansion, the more frequently occurring strings will be dynamically swapped into the short dictionary. The dictionaries are used as scratch pads during the time of program execution and need not be stored/transmitted otherwise.

A damaged compressed file with few redundancy can hardly be deciphered, if not impossible. By using the idea of Efficient Dispersal of Information originated by Dr. Michael O. Rabin of Harvard University, a practical implementation of the algorithm has been designed. By manipulating a file F of length L into n subfiles F_i, $1 \leq i \leq n$, the length of each F_i is L/m where m is an integer smaller than n. If no more than n-m subfiles have been damaged, any remaining m subfiles can be used to reconstruct the original file F quite easily.

Acknowledgments

First, I wish to thank Professor Jacob Rootenberg, who unfortunately passed away in May 1991, for originally arousing my interest in data compression and for his generous advice, encouragement. Acknowledgment is also due to my present mentor, Professor Charles Giardina, for taking me as one of his many students when my second mentor suddenly left the University at the beginning of the fall semester 1992. Thanks also go to him for his stimulating data compression course in the fall semester 1991, and Professor Michael Anshel for his interesting course of symbolic and algebraic manipulation systems in the spring semester 1992.

Finally, thanks go to my wife Irene for her understanding and support, daughters Olivia Yan-Hui and Alice who was born during the preparation of this thesis for delightfully distracting me from work.

Table of Contents

List of Tables

List of Figures

1 Introduction

The thesis consists mainly of two parts: a new string pattern matching algorithm and a new lossless data compression algorithm which also includes a practical implementation of Information Dispersion Algorithm for fault-tolerance.

String pattern matching algorithms are widely used in many areas, such as text editing, information retrieval and lossless data compression. Many word processors have a built-in string pattern matching algorithm for string searching.

Data compression becomes more and more important and practical nowadays because we need to store and transmit a large amount of data and the power of computers becomes more and more cheaply available.

Some files, such as text and computer object files, have to be reconstructed exactly from their compressed form. This kind of compression is called **lossless**.

Other files, such as audio and video files, can also be compressed by **lossy** algorithms because the exact replica of the original files are not necessary as long as the audio and video effects are acceptable or tolerable to the human ears and eyes. For these kinds of files, lossy algorithms instead of lossless ones are often used because a much greater compression ratio is needed and can be achieved because image or video needs so much storage to store or so much bandwidth to transmit.

2 String Pattern Matching

A string pattern matching algorithm is to use a character string, ***pattern***, to search another character string, ***text***, for the first or all occurrence(s) of the *pattern* in the *text*.

Suppose that *pattern* is a string of length ***patlen***, *text* is a string of length ***textlen***, (*patlen* is usually much smaller than *textlen*) and we wish to find the location *i* of the leftmost character of a substring of the *text*, which matches the *pattern*:

```
pattern:          DAD-DAD
text:             OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                  ↑
```

Fig. 1 Sample *pattern* and *text* strings

2.1 Straight Forward (SF) Algorithm

The straight forward (SF) or naive approach of course comes to our mind. It aligns the *pattern* with the *text* at the leftmost position, it compares *pattern* and *text* one character by one character from left to right. If any character does not match, it shifts the *pattern* one position to the right; although in practice, only a pointer or index needs to be changed. The procedure is repeated until it exhausts the *text*, in that case it finds no match, or finds at some position *i* that all characters of the *pattern* match a substring of the *text*. The worst case time complexity of SF is quadratic. Suppose that a *pattern* is $a^m b$ and a *text* is a^n (where $n > m$). The *pattern* starts at the leftmost position, and we have m successes and a failure; then we shift the *pattern* one position to the right, and we also have m successes and a failure. This procedure executes total (n-m) times, each

time (m+1) comparisons are made, and finally we find no match. The time needed is proportional to (m+1)*(n-m) = m*n + other smaller terms. Hence we say that the time complexity of SF is m*n in the worst case. In practice, such cases are extremely rare, and SF usually has a better chance of mismatch than a match for each comparison. Hence SF is almost linear in (*i* + *patlen*) [Smit].

2.2 Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt (**KMP**) [Knuth77] have described a pattern matching algorithm which has linear time complexity of (*i* + *patlen*) in the worst case. In practice, its performance is nevertheless about the same as that of SF [Smit].

Initially, the KMP algorithm aligns the *pattern* with the *text* at the leftmost positions, it then compares *pattern* and *text* from left to right. Based on the result of comparison, it will go from a state to another state of the automaton which has been built during the process of preprocessing the *pattern*. In order to reduce the number of edges coming out of each state, it uses only two kinds of links: success link and failure link. Each state of an automaton contains a *pattern* symbol with which the corresponding *text* character is to be compared. The automaton consists of an initial state, a final accepting state and other states corresponding to the *pattern* characters. A semaphore called "Success" or "S" for short is set in the initial state. The semaphore S will also be set when *pattern* and *text* characters match, otherwise the "Failure" or "F" semaphore will be set. If the S semaphore is set, a *text* character will be read when the automaton go from a state to the next higher state along the S link. When the current *text* character β

does not match the corresponding *pattern* character γ, the automaton will go to another state along an F link so that it will try to shift the *pattern* to the right such that the longest suffix α of the *text* characters at the left side of the current *text* character β will match the corresponding *pattern* characters and the *pattern* character corresponding to β is not equal to γ. The following example will illustrate how Fail[4] link is calculated if we have three *pattern* characters and corresponding *text* characters matched and we are now comparing the 4th *pattern* character with its corresponding *text* character.

We use *j* to order the following example *pattern* positions:

```
j:                                  1234567
pattern:                            DAD-DAD
text:                          ...  DADA  ...
pattern to be shifted to:             DAD-DAD
```

Hence Fail[4] = 2 since we can reuse the 1st *pattern* character 'D' and compare the 2nd *pattern* character with the current *text* character.

Other Fail links can be similarly calculated. The automaton for the above *pattern* is constructed as shown in Fig. 2. Improvements over the F links are shown as "Next" or "N" links in Fig. 3.

2.3 Boyer_moore (BM) Algorithm

Boyer-Moore (BM) [Boyer] have developed a revolutionary pattern matching algorithm. The BM algorithm first aligns the leftmost *pattern* character with the leftmost character of the *text*. But it compares the rightmost character of the *pattern* first. If it is a mismatch and the *text* character is not in the *pattern*, the whole *pattern* of length *patlen*

can be shifted to the right. In the example in Fig. 1, when we compare the rightmost 'D' of the *pattern* with the '4' of the *text*, we know that '4' is not one of the *pattern* characters, so we can shift the whole *pattern* to the right:

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                          ↑
```

In this case, we can skip six comparisons. A great achievement! In practice, its performance is significantly faster than that of SF and KMP approaches [Smit].

The BM algorithm calculates two heuristics, called delta1 and delta2 for moving pointers to the *pattern* and the *text*. For each current *text* character β in the alphabet, if β is not one of the characters of the *pattern*, delta1[β] is set to *patlen*, else align the *pattern* so that β will match the same rightmost character in the *pattern*, then the distance between the same β character in the *pattern* and the rightmost character of the *pattern* is the value of delta1[β]. Note that the BM algorithm compares the *pattern* and the *text* from right to left starting from the rightmost character of the *pattern*. Suppose we have a substring α of the *text* matching the corresponding characters of the *pattern*, we are now comparing the next left *text* character β (β is the left neighbor of α in the *text*) with its corresponding *pattern* character γ, if β matches γ, we then form $\beta\alpha$ to be our new α, get our next left character of the *text* to be our new β and repeat the procedure; else we shift *pattern* to the right at least one position until α matches the corresponding *pattern* characters and the *pattern* character corresponding to β is not γ, or the longest suffix of α matches a prefix of the *pattern*. Delta2 value is the distance between β and the rightmost character of the *pattern* at the new location. Actually, delta1 and delta2

tables are calculated based on the given *pattern* only, without any information of the *text*. We use the following *patterns* "DAD-DAD" and "DAD-AAD" to illustrate how to calculate delta1[β] for β = '-' and delta2[j] for j = 5 for these two *patterns*, respectively.

For calculating delta1['-'], we use *j* to order the example *pattern* positions from left to right:

```
j:                                  1234567
pattern:                            DAD-DAD
text:                          ...      -   ...
pattern after aligning '-':          DAD-DAD
pointer i:                              ↑  ↑
```

From the last line, pointer *i* moves from the left arrow to the right arrow, since the distance between these two arrows is 3, hence delta1['-'] = 3.

For calculating delta2[5], we have two rightmost *pattern* characters and their corresponding *text* characters matched, and we are now comparing the 5th *pattern* character.

```
j:                      1234567
pattern:                DAD-AAD (This is a different pattern)
text:                       -AD
pattern after aligning α: DAD-AAD
pointer i:                  ↑     ↑
```

From the last line, pointer *i* moves from the left arrow to the right arrow, since the distance between these two arrows is 6, hence delta2[5] = 6.

The complete delta1 and delta2 values for these two patterns of "DAD-AAD" and

"DAD-DAD" are shown in Fig. 4 in the appendix.

The BM algorithm uses the maximum value of delta1 and delta2 to move pointer *i* to the *text* and the rightmost *pattern* character will align with pointer *i* whenever pointer *i* moves to the right. As in KMP algorithm, the delta1 and delta2 values are calculated based on the given pattern only, not depending any information of the *text*.

2.4 Proposed Solution

A new string pattern matching algorithm presented here consists of two parts. The first part is to form a **mapping table** of the size of the alphabet, i.e., a function from distinct characters of the *pattern* to successive positive integers; and the same characters in the *pattern* are linked together. It then builds an automaton such that the best *jump* and the next state is calculated given each input *pattern* character and each state. The number of states of the automaton is equal to the length of the *pattern*, and the number of columns of the automaton is equal to one plus the number of distinct characters in the *pattern*. The second part is to perform the search operation. Its search loop consists of only three statements. It uses the mapping table and the automaton, which have been formed in the first part, to look up for *jump* and the next state, and just add *jump* to the pointer pointing to the *text*.

2.5 Mapping Table and Automaton Construction

First, an array of the size of the alphabet is needed. For byte-addressable computers, an array k[256] can be a good choice, because each combination of the 8 bits

is possible.[1] We use *j* to order *pattern* positions from right to left, and use indices under the *pattern* for the convenience of discussion:

```
j:          7654321
pattern:    DAD-DAD
indices:    423 211
```

For the same characters we write indices underneath the *pattern* for the purpose of discussion. We have two 'A', hence we write 1, 2 under these 'A' from right to left; for the four 'D', we write 1, 2, 3, 4 under these 'D', we have only one '-', so we don't need to put any index under '-'. Indices will sometimes also be used for the *text* whenever we need to differentiate the same characters at different positions. We will also use subscripts to represent these indices. Assume ASCII code is adopted, then the ASCII codes for all the distinct characters of the *pattern*, i.e., 'D', 'A' and '-', are as follows:

Order	3	2	1
Character	-	A	D
Code in decimal	45	65	68

Table 1 ASCII Code Table for the *pattern* 'DAD-DAD'

We process the *pattern* from right to left, and use *j* to number them from right to left starting from one. The procedure for forming a mapping table of a *pattern* is as follows:

For each distinct character in the *pattern* from right to left, we use the ASCII

[1]For some languages, such as Chinese, each character of the alphabet needs two bytes for storage, an array k[65536] should be used for the mapping table.

code of the character, *ascii-code*, as an index to the array cell *k*[*ascii-code*] where we put a successive positive integer starting from one.

The entries of the array *k*[*ascii-code*] are initially set to zeros. For example, when we get the first 'D', i.e., D_1, we check k['D'], i.e., k[68], it is zero, so we set k[68] = 1, the first positive integer, and input_link[1] = 1 where the index of input_link is qual to the number of distinct characters seen so far, and the 1 at the right side of the equal sign means the next state of the automaton after seeing a character the automaton expects or position *j* of the current *pattern* character. So, when we see an input character 'D' at state zero we will go to state 1 of the automaton. Then we get to the second character, i.e., A_1, we have k['A'] = k[65] = 2, the next positive integer, input_link[2] = 2. For D_2, we find that k['D'] is not zero, we leave k['D'] alone, but use another pointer state_link[1] = 3 to link D_1 and D_2 of the *pattern* because the positions *j*'s of these D's are 1 and 3, respectively. For '-', we have k['-'] = k[45]= 3, the next successive positive integer, input_link[3] = 4 because 4 is the position *j* of '-' in the *pattern*. For D_3, we get state_link[3] = 5 to link D_2 and D_3. For A_2 we have state_link[2] = 6 to link A_1 and A_2. For D_4, we have state_link[5] = 7 to link D_3 and D_4. So the same characters in the *pattern* are chained together.

After the whole *pattern* has been inspected, we have formed three items: a mapping table, an array *input_link* to link each distinct input *pattern* character to its *pattern* position *j*, and an array *state_link* to chain the same characters in the *pattern*, in Tables 2, 3 and 4, respectively.

	'-' 45			'A' 65		'D' 68	
0 ...	3	0 ...		2	0 ...	1	0 ...

Table 2 Mapping Table k[256] for the *pattern* 'DAD-DAD'

Order	3	2	1
Input character	'-'	'A'	'D'
input_link	4	2	1

Table 3 input_link Table for linking distinct *pattern* characters to *pattern* positions

state_link	0	0	7	0	5	6	3	0
Automaton state no. after seeing the char	7	6	5	4	3	2	1	0
	D	A	D	-	D	A	D	#

Table 4 state_link Table for linking the same characters,
where # denotes any other character not in the *pattern*.
In practice, state 7 is not needed since it is a success state.

The above two arrays in Tables 3 and 4 are used to link the same characters together so that when we are given a mapping table code of an input character, we want to see if the character will be one of the several same characters at different positions of the *pattern* so that the automaton can calculate the longest jump. The data structure relationship among Tables 2, 3 and 4 is shown as in Fig. 5 in the appendix.

Now, we wish to construct the automaton itself. First we give examples to illustrate how the automaton works and then give a general procedure.

Initially, we align the *pattern* and the *text* at their leftmost characters, and the

automaton is in state zero. We compare the *pattern* and the *text* from right to left, starting from the rightmost character of the *pattern*. Suppose the automaton is now in state zero and the current *text* character is '4', then k['4'] will be zero since this character is not in the *pattern*, so we shift the whole *pattern* to the right of the character '4'. Shifting anything less than *patlen* will not do any better. Therefore we set *jump* (or ***action***) for the automaton a[0][k['4']] = a[0][0] = *patlen* = 7, as shown in the following diagram. In practice, we only need to move pointer *i* associated with the current character of the *text* seven positions to the right.

```
j:                    7654321
pattern:              DAD-DAD
text:           ... 4 ...
```

Next, suppose the current *text* character is 'A' and aligns with D_1, as shown in the following diagram:

```
j:                    7654321
pattern:              DAD-DAD
text:                  ...  A  ...
```

In this case, the character 'A' of the *text* does not match the *pattern* character, but we can find the nearest substring which matches 'A' and is at the left side of 'A', i.e., A_1. Hence, we can align 'A' with A_1 for the next comparison. k['A'] is found from the mapping table to be 2. Therefore, a[0][k['A']] = a[0][2] = 1, the distance between A_1 and D_1.

Now, assume we have the following snap shot:

```
j:                          7654321
pattern:                    DAD-DAD
text:                    ...  NAD  ...
pattern after shifting:          DAD-DAD
```

Ignore the last line for the moment. We have the first two characters of the *pattern* and the *text* at the right to be same, but the third character of the *text*, i.e., 'N', is not in the *pattern*, then for substring 'AD' we try to find its longest suffix which can match a prefix of the *pattern*. In this case, 'D' is the answer. Therefore, we have to align 'D' with D_4, as shown in the last line of the above diagram, hence we shift the *pattern* to the right six characters. Since k['N'] = 0 and we have already matched two characters (that means the automaton is in state 2,) we should set a[2][k['N']] = a[2][0] = the distance between 'N' and the new D_1 (i.e., the rightmost 'D' as shown in the last line of the above diagram) = 8.

Now, assume we have the following case:

```
j:                          7654321
pattern:                    DAD-DAA  (a different pattern)
text:                    ... DDAA  ...
pattern after shifting:          DAD-DAA
```

In this case, we find that the first three characters match each other between the *pattern* and the *text*, but character D_2 of the *text* doest not match character '-' in the *pattern*. First we try to find whether 'DDAA' is a substring of the *pattern* at its left side. No, it is not. Then we try to find the longest suffix of 'DDAA' to match a prefix of the *pattern*. It also fails. Therefore, we have to shift the whole *pattern* to its right. Shifting anything

less than *patlen* will not do any better.

Now, assume we have the following case which was shown before to calculate Boyer-Moore's delta2 value to be 6:

```
j:                          7654321
pattern:                    DAD-AAD  (a different pattern)
text:                     ...  -AD  ...
pattern after shifting:           DAD-AAD
```

In this case, we find that the first two characters match each other between the *pattern* and the *text*, but character '-' of the *text* doest not match character A_2 in the *pattern*. First we try to find whether "-AD" is a substring of the *pattern* at its left side. No, it is not. Then we try to find the longest suffix of "-AD" to match a prefix of the *pattern*. D_3 is the longest such substring. Therefore, we have to shift the *pattern* so that D_3 of *pattern* is aligned with 'D' of the *text*. The distance between '-' of the *text* and the rightmost character of the *pattern* at the new location is 8, which is greater than Boyer-Moore's delta2 value of 6.

In summary, assume that we have already matched a substring β of m characters, (m = 0 is just a special case, i.e., β is an empty substring, that also means we just start to do the comparison at the rightmost character of the *pattern*) and we are now considering the current *text* character α associated with pointer i, we have noticed the following rules in order to get the longest *jump* (or best *action*):

To calculate a *jump*, all we need is to align the *pattern* with the *text* such that the *pattern* is moved as far to the right as possibe, but also we don't want to miss any occurence of the *pattern* inside the *text*; then the distance between pointer to the *text* and the rightmost

character of the *pattern* is our *action* value.

1. If α is not in the *pattern*, we try to find the longest suffix γ of β that matches a prefix of the *pattern*. Align γ with this prefix. (If there is no such non-empty γ, the length of γ is just zero, a special case.)
2. If α is in the *pattern*

   then

   {

   if α matches its aligned *pattern* character,

   then *action* = -1, i.e., we move *i* to the left one position;

   else

   {

   if $\alpha\beta$ matches a substring of the *pattern* that is at its nearest left side, then we align $\alpha\beta$ with that substring;

   else we try to find the longest suffix γ of β that matches a prefix of the *pattern*. Align γ with this prefix. (empty γ is just a special case)

   }

   }

We can also get other values for the *action* table entries of the automaton. The calculation of the state table of the automaton is quite easy, if *action* is a negative one then its corresponding state entry will be equal to its next higher state.

The final automaton for the example *pattern* can be obtained as follows:

		Input characters			
input_link →		0	1	2	4
state_link ↓	states ↓	#	D	A	-
		0	1	2	3
0	0	7	-1	1	3
3	1	7	7	-1	3
6	2	8	-1	8	8
5	3	7	7	7	-1
0	4	8	-1	8	8
7	5	9	9	-1	9
0	6	10	-1	10	10

Table 5 Action Table of the automaton for the *pattern* 'DAD-DAD', where # represents any character not in the *pattern*.

		Input characters			
input_link →		0	1	2	4
state_link ↓	states ↓	#	D	A	-
		0	1	2	3
0	0	0	1	0	0
3	1	0	0	2	0
6	2	0	3	0	0
5	3	0	0	0	4
0	4	0	5	0	0
7	5	0	0	6	0
0	6	0	7	0	0

Table 6 State Table of the automaton for the *pattern* 'DAD-DAD'

2.6 The Algorithm

2.6.1 Preprocessing a Pattern

Now, we give a general procedure in pseudo code to calculate action table and state table entries of the automaton:

Due to the importance of readability, I don't follow any programming language 100%, but the style is more inclined towards the C programming language. Sometimes plain English is used instead to describe a procedure. Unless stated otherwise, all indices of arrays start from zero, following the C program convention, and all array entries are initialized to zero. Let *column* denote the number of distinct characters of the *pattern*, and β denote the substring of the matched characters so far and *len* denote the length of

a suffix of β currently under consideration, and we are currently comparing a *text* character α with its aligned *pattern* character.

```
a[0][0] = patlen;
a[0][1] = -1;
for (j = 2; j <= column; j++)   /* "j++" means "j = j + 1" in C */
        {
/* The remaining entries of the first row of the automaton, i.e., i = 0, state zero */
        link = input_link[j];
        a[0][j] = link - 1;
        }
prefix = 0;
for (i = 1; i < patlen; i++)        /* i: state of the automaton */
        {
        /* The remaining entries of first column of the automaton, i.e., j = 0 */
        /* For j = 0, the current character of the text, α, is not in the pattern */
        len = i;
        if β does not match a prefix of the pattern, then len = prefix;
        a[i][0] = i - len + patlen;
        prefix = len; /* save len for future use */
        }
for (i = 1; i < patlen; i++)        /* i: state of automaton */
        {
```

```
for (j = 1; j <= column; j++)
        /* column: no. of distinct characters of pattern */
        {
        /* α is in the pattern */
        for (link = input_link[j]; link > 0; link = state_link[link])
                {
                if (link > i + 1)
                        {
        /* α already matches the pattern character at pattern position link */
                        if β matches the substring of pattern starting from pattern
                        position (link - 1)
                                {
                                a[i][j] = link - 1;
                                goto j_loop;
                                }
                        }
                else if (link == i + 1)
                        {
                        /* the pattern character at pattern position link is the
                        character that the automaton at state i expects */
                        a[i][j] = -1;
                        s[i][j] = link;
```

```
                    goto j_loop;
                    }
                }       /* link loop */
            a[i][j] = a[i][0];
j_loop:     continue;
            }       /* j loop */
        }       /* i loop */
```

2.6.2 The Search Part of the Algorithm

The search part of the algorithm is very simple. The search loop consists of only three statements: Fetch an *action* and a next state from the automaton by table look up. Then add *action* to the current pointer *i* associated with the *text*. The *pattern* itself is not used here at all and the search part of the algorithm in pseudo code is as follows:

```
state = 0;      /* Initially, the automaton is in state 0 */
i = patlen - 1; /* Index i points to the patlenth character of the text */
while (not end of text file and state != patlen) do
    /* "!=" means not equal in C */
    {
    j = k[text[i]];             /* text[i] is the current text character */
    i = i + a[state][j];
    state = s[state][j];
    }
```

```
if (state == patlen)  /* The automaton is in the success state, match found */
        print(i + 2);  /* i + 2 is the starting position of the match starting from 1 of 1st
                          text character */
else print("no match");
```

2.7 Example

Now, we use an example to demonstrate the simplicity of the search part of the algorithm. Note that from a state number of the automaton we know how the *pattern* is aligned with the *text;* therefore, we need not concern ourselves with any index pointing to the *pattern*. Actually, we don't need the *pattern* at all in the search loop, the *pattern* shown in each snap shot below is only used for the purpose of illustration.

For the same *pattern* and *text* as before, the mapping table, the automaton action and state tables have been formed in Table 2, 5 and 6, respectively by the preprocessing part of the algorithm.

Initially, the automaton state is zero, and we align the *pattern* and the *text* at their leftmost characters.

```
pattern:            DAD-DAD
text:               OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                  ↑
```

First, i = *patlen* - 1 = 6 (Note that we start counting from zero in the program), from the action table we can find a[0][k['4']] = a[0][0] = 7, so we shift i to the right 7 characters, i.e., $i = i + 7 = 6 + 7 = 13$. From the state table, we can find s[0][0] = 0. So we have the following picture:

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                      ↑
```

Then, a[0][k['A']] = a[0][2] = 1, s[0][2] = 0. So, we move the pointer *i* one character to the right, i.e., $i = i + 1 = 13 + 1 = 14$. Now we have

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                      ↑
```

Then, a[0][k['D']] = a[0][1] = -1, we move the pointer to the left one character; and s[0][1] = 1. $i = i + (-1) = 14 + (-1) = 13$.

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                     ↑
```

We have a[1][k['A']] = a[1][2] = -1. We move the pointer *i* to the left one character, i.e., $i = i + (-1) = 13 + (-1) = 12$; and s[1][2] = 2.

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                     ↑
```

Now, a[2][k['N']] = a[2][0] = 8. We move the pointer *i* to the right eight characters, i.e., $i = i + 8 = 12 + 8 = 20$; and s[2][0] = 0.

```
pattern:                    DAD-DAD
text:           OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                      ↑
```

We have a[0][k['D']] = a[0][1] = -1. We then move the pointer *i* to the left one

character, i.e., $i = i + (-1) = 20 + (-1) = 19$; and s[0][1] = 1.

```
pattern:                              DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                 ↑
```

We have a[1][k['A']] = a[1][2] = -1. We then move the pointer i to the left one character, i.e., $i = i + (-1) = 19 + (-1) = 18$; and s[1][2] = 2.

```
pattern:                              DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                 ↑
```

We have a[2][k['D']] = a[2][1] = -1. We then move the pointer i to the left one character, i.e., $i = i + (-1) = 18 + (-1) = 17$; and s[2][1] = 3.

```
pattern:                              DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                 ↑
```

We have a[3][k['I']] = a[3][0] = 7. We then move the pointer i to the right 7 characters, i.e., $i = i + 7 = 17 + 7 = 24$; and s[3][0] = 0.

Now, we have

```
pattern:                              DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                    ↑
```

Then, a[0][k['D']] = a[0][1] = -1, $i = i + (-1) = 24 + (-1) = 23$ and s[0][1] = 1.

```
pattern:                              DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                 ↑
```

Then, a[1][k['A']] = a[1][2] = -1, $i = i + (-1) = 23 + (-1) = 22$ and s[1][2] = 2.

```
pattern:                                DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                  ↑
```

Then, a[2][k['D']] = a[2][1] = -1, $i = i + (-1) = 22 + (-1) = 21$; and s[2][1] = 3.

```
pattern:                                DAD-DAD
text:                 OLIVIA4-14YANADHUIDAD-DAD MISSES YOU ...
pointer i:                                  ↑
```

Then, a[3][k['-']] = a[3][3] = -1, $i = i + (-1) = 21 + (-1) = 20$; and s[3][3] = 4.

For the next three characters we can have the following three lines:

a[4][k['D']] = a[4][1] = -1, $i = i + (-1) = 20 - 1 = 19$, s[4][1] = 5.

a[5][k['A']] = a[5][2] = -1, $i = i + (-1) = 19 - 1 = 18$, s[5][2] = 6.

a[6][k['D']] = a[6][1] = -1, $i = i + (-1) = 18 - 1 = 17$, s[6][1] = 7.

State 7 is a success state! Hence we have found a match. We have to adjust i because now i points to a character at the left side of the matched string. Hence we add 2 to i to get the result of 19 starting from 1 of the 1st *text* character.

We can see that during the search process, the state of the automaton will take care of the alignment with the *pattern* automatically. We need to move only one pointer, i.e., pointer i associated with the *text*.

2.8 Experimental Results

In order to obtain a fair comparison between the algorithm and the Boyer-Moore algorithm, the two have been coded in C, compiled using the same compiler options and

run using the same computer system. They read the same *patterns* and *text*, and produced the same amount of output characters. One of the author's papers [Chang91B] has been used as a *text* data file. The file is a WordPerfect file converted to ASCII file of 28249 bytes long.

Two experiments have been conducted. The first one was to determine a ratio of a sum of both *pattern* and *text* characters inspected versus a number of *text* characters passed. As we said before, the algorithm doesn't need to inspect *pattern* characters during the process of searching the *text*. For each algorithm eight different *patterns* were chosen, and *pattern* lengths of from one to 14 were used for testing, same as in Boyer-Moore's paper. The results of the eight patterns were averaged for the same pattern length from one to fourteen, and are shown as in Fig. 6 in the appendix. The results confirm our claim that the algorithm has a better ratio of the number of characters inspected over the number of characters passed for all lengths except *pattern* length of one, and the ratio for *pattern* length of one is the same for the two algorithms.

The second experiment was to determine an overall performance for each of the algorithms. The results of a particular *pattern* of length 60, from which prefixes of length from one up to thirty with an increment of one, thereafter with increments of five up to 60 were tested, are shown as in Fig. 7 in the appendix. Each data point is an average value of ten tests. All *patterns* match the *text* at position *i* of 26153 except that the *pattern* whose *patlen* (the length of *pattern*) is equal to one matches the *text* at position *i* of 1574. It shows that overall performance of the algorithm is better than that of BM's for pattern lengths of not longer than about 25, a reasonably long pattern, whereas for

pattern lengths of longer than 25 the latter is better. The ratio of the time spent in the algorithm versus that of the Boyer-Moore algorithm is nearly linear except at the *patlen* equal to 1; but its penetration length is very different from the others. By calculating a parameter $\rho = i / patlen^2$ for each data point, we have found ρ equal to about 41 when the time ratio is equal to one. Hence, we suggest that readers should use the algorithm instead of the Boyer-Moore algorithm when ρ is greater than 40, otherwise the latter should be used for better performance.

Only case sensitive string matching was compared in the experiments. Obviously, the algorithm would have a performance boost over its counterpart for non case sensitive string matching.

2.9 Extensions

2.9.1 Non Case Sensitive Pattern Matching

Non case sensitive languages, such as Fortran, treat same letters of different cases as same ones. They can benefit from the algorithm enormously. The algorithm only needs to map same letters of different cases to a same number in a mapping table. For example, we can build a non case sensitive mapping table from the given *pattern* 'DAD-DAD' as follows:

	'-'		'A'		'D'		'a'		'd'	
	45		65		68		97		100	
0 ...	3	0 ...	2	0 ...	1	0 ...	2	0 ...	1	0 ...

Table 7 Mapping Table k[256] for the non case sensitive *pattern* 'DAD_DAD'

There is no change to be made to the automaton and the search part. The search loop does not need to do any case conversion now.

2.9.2 Pattern Matching with Don't Care Symbols

String pattern matching with don't care symbols can be implemented quite simply by the algorithm as follows:

In building an automaton during preprocessing a pattern, a don't care symbol in a pattern matches any character. If i is the pattern position of the don't care symbol, then action entries at state i - 1 are set to -1 for the whole row, and the state entries at state i - 1 are set to i for the whole row. The search part is the same as before.

2.9.3 Pattern Matching with a Group of Characters

String pattern matching with a group of characters can also be implemented by just modifying the algorithm part of preprocessing the pattern. If every character of the group doesn't appears anywhere else (but the group may repeatedly occur in the pattern,) then every character of the group will map to a same number in the mapping table, similar to the non case sensitive pattern matching. If any character in a group appears elsewhere in the pattern but these two characters don't belong to two identical groups, then this character is considered as an element of the group in that pattern position, and in the program we have to check all these elements for that pattern position to find the longest jump. The search part is the same as before.

2.9.4 Pattern Matching with a Complement of a Group of Characters

String pattern matching with a complement of a group of characters can also be implemented by just modifying the algorithm part of preprocessing the pattern. We can just include the characters of the complement of the group of characters for that pattern position, and the problem is just pattern matching with a group of characters.

2.10 Improvements

Experimental results showed that a comparison of the rightmost *pattern* character with a *text* character had over 90% chances of mismatches. The search loop can be further modified as shown below to yield a higher performance. In state zero of an automaton we don't need to calculate the next state of the automaton as long as there is no match. Also a *pattern* is appended to the end of the *text* in the work space in order to eliminate the checking of the end of *text* file in the while loop. The modified search loop is shown as follows:

```
for (i = 0; i < patlen; i++) text[textlen + i] = pattern[i];
i = patlen - 1;
l_search:
do      {
        action = a[0][k[text[i]]];
        i += action;
        } while (action > 0);
state = 1;
```

```
while (state != patlen)
        {
        action = a[state][k[text[i]]];
        i += action;
        if (action > 0) goto l_search;
        else state++;
        }
if (i < textlen - patlen) /* Match found before the appendix of pattern to the text */
        print(i + 2);  /* i + 2 is the starting position of the match starting from 1 of 1st
                          text character */
else print("no match");
```

As you can see from the above improved version, the state table of the automaton is no longer needed now because either a state value of zero is implied, or it can be simply set or incremented.

Performance can be further improved by using a composite mapping table of ak[j] = a[0][k[j]] for the first row of a[][], so that the above do-while loop can be made faster.

2.11 Space and Time Complexity

2.11.1 Storage Requirement

Besides the input buffers for the input *pattern* and *text*, we need the following storage for the algorithm:

mapping table k[]: 256 bytes

input_link[]: *patlen* bytes (only needed during preprocessing the pattern)

state_link[]: *patlen* bytes (only needed during preprocessing the pattern)

automaton action table a[][]: *patlen* × *patlen* bytes

automaton state table s[][]: *patlen* × *patlen* bytes, this table is not needed if an improved version of the algorithm is used.

Since the length of a pattern, *patlen*, is usually short, excluding the space required by the *text*, the storage required in the algorithm is quite small.

2.11.2 Time Complexity

2.11.2.1 Preprocessing a *Pattern*

It is easy to see that the time required to build a mapping table, input_link array and state_link array is linear in time of (*patlen* + 256).

1. For the case of a pattern which has all characters different from each other, then all state_link entries will be zero, hence the link loop will execute only once, *column* is now equal to *patlen* because all characters of the *pattern* are different. Therefore, the time due to the main loop which contains the nested i loop, j loop and link loop is in the order of $patlen^2$. Obviously the other parts of the algorithm have time complexity not larger than $patlen^2$. Hence the time complexity of preprocessing a *pattern* of different characters is $patlen^2$.

2. For the case of a *pattern* whose characters are all same, then *column* is equal to 1. Each *link* loop will execute 2 * *i* number of times: First we have to propagate from the right end of the *pattern* to the current position *link* such that *link* == i + 1

(from 1 to *link*), secondly we will compare characters from position *link* to the right end of the *pattern*. Hence the time of executing each link loop is equal to 2 * i. Considering the i loop and j loop (j loop only executes once) also, we get the total time = 2 + 4 + 6 + ... + 2 * i + ... + 2 * *patlen* = *patlen*(*patlen* + 1) = $patlen^2$ + a smaller term. Other parts of the algorithm have a complexity not larger than this number. Hence the worst case time complexity of preprocessing a *pattern* of same characters is $patlen^2$.

3. For a general *pattern* our conjecture is that the worst case time complexity for preprocessing a *pattern* is $patlen^2$.

2.11.2.2 Searching a *Text*

Since the Boyer-Moore algorithm has been shown that the time complexity for the worst case is linear in time of (*textlen* + *patlen*) [Guiba]. The worst case time complexity of searching a *text* for the algorithm is conjectured to be also linear in time of (*textlen* + *patlen*) since the jumps in the search loop of the algorithm are at least as good as that of the BM algorithm.

2.12 Summary

Having a better ratio of the number of characters inspected over the number of characters passed than that of the Boyer-Moore algorithm during the search operation, the algorithm also has a simple search loop, hence it competes favorably with the BM algorithm. For a current *text* character α not matching its corresponding *pattern* character μ, the BM algorithm uses only the partial information of α (it uses only the fact that α

is not equal to μ) and its right neighbor, the matched *pattern* character substring β, for calculating a *delta2* value similar to our *jump* whereas the algorithm uses the full information of both α and β to match another possible substring of the *pattern* to find a *jump* value. Hence *jump* is always greater or equal to the maximum of *delta1* and *delta2* in the BM algorithm. The algorithm can be used for searching a non case sensitive pattern or pattern with don't care symbols and a group of symbols, very effectively. The drawback of the algorithm is that more time for preprocessing a *pattern* will be needed compared with the BM algorithm. Because the length of a *pattern* is usually much smaller than that of a *text*, the more time spent in preprocessing the *pattern* is worth the effort to speed up the search loop in practice for reasonably short *pattern*. We suggest the readers should consider using the algorithm instead of the B-M algorithm if the value of $i/patlen^2$ is greater than 40, where i is the position of the substring of the *text* which matches the *pattern* and *patlen* is the length of the *pattern*.

Like the BM algorithm, the algorithm has to move a pointer both forward and backward, which may cause I/O buffer complication, whereas the KMP algorithm has the advantage of no backup being needed because it moves the pointer associated with *text* either forward or does not move it, but it will never move it backward. Because if a *text* is too large to be read all into memory once, then only a part of it can be read into I/O buffer. In this case, if a pointer has to be moved beyond the current buffer, and it also has to be moved backward later, then the I/O buffer management is complicated, and the program for the algorithm has to do more work.

The algorithm matches only one *pattern* with a *text* in one pass. If a group of

pattern strings are to match a *text* in one pass, the method which has been developed by Aho and Corasick should be considered [Aho75].

3 Data compression

3.1 Lossy Data compression

3.1.1 Image Transform

Transforms have been very important in many scientific fields. It is well known that before calculators were widely available, in order to get the multiplication product of two numbers x and y, we could first find the logarithms of these two numbers $\log_{10}x$ and $\log_{10}y$ from a logarithmic table, add these numbers by hand, then find the inverse of the addition from an antilogarithmic table, the final result was the product we wanted. The same is true for representing images. For example, we can digitize a still picture by uniformly sampling it along two rectangular coordinates. Each sample is a picture element (pixel), and we can use a quantized number to represent the gray level of a black and white pixel or the color of a color pixel. For each color pixel, 8-bits, 15-bits, 16-bits and 24-bits can display 256, 32,768, 65,536 and 16,777,216 colors, respectively. The IBM 8514/A video standard has the resolution of 1024 x 768, each pixel has 8 bits for a total of 256 colors. Just a picture of this resolution, which is worse than analog pictures, would need 786,432 bytes of storage. A movie at 30 frames per second would require much more storage to store or much more bandwidth to transmit. Data compression and other digital signal processing techniques can be more effectively performed by transforming digital image in spatial domain into another domain, say frequency domain.

3.1.2 The JPEG Still Picture Compression Standard

The standard for compressing still pictures [Walla] is being worked by a society by the name of Joint Photographic Expert Group (**JPEG**). After reviewing 12 competing compression algorithms, JPEG determined that Discrete Cosine Transform (**DCT**) was the best algorithm. DCT is also being adopted by Motion Picture Experts Group [LeGal] and px64 kbit/s video coding standard [Liou].

The JPEG still picture compression standard is currently widely used in the industry. Among them are Adobe's PostScript page description language for printing systems, and the future CCITT color facsimile standard. The MPEG motion picture compression standard and px64 kbit/s video coding standard are also widely accepted by the industry.

3.2 Lossless Compression

Currently, there are three major lossless compression methods: Huffman [Lelew, Welch], Arithmetic [Clear, Witte] and Lempel-Ziv [Mille, Welch, Ziv, Ziv77, Ziv78] codings. For compression coding methods with detailed descriptions see [Bell, Lelew]. A summary of the above three compression methods is given below:

3.2.1 Huffman Coding

In Huffman coding, the mapping of input source symbol to output code is fixed and can be formulated in a table. To minimize the redundancy, Huffman coding assigns a short bit string code for a more frequently occurring input symbol or bit string, a long

bit string code for a less frequently occurring input symbol or bit string in the following algorithm. Initially, a group consists of a union of n elements of an alphabet. These elements a_i, $1 \leq i \leq n$, have known probability occurrences $P(a_i)$ based on some statistics, and are arranged in a non-increasing order of probability of occurrences. Huffman algorithm chooses two symbols of the smallest probabilities, form a node for each symbol and create a third node to be their father and its probability is assigned to the sum of probabilities of his sons. Now, exclude these two sons from the group, and add their father to the group in the proper place so that the non-increasing order of the probability of occurrences is preserved. The above process is iterated until the group has one element and it is the root of the tree just formed. The leaves of this binary tree are the original n symbols. The Huffman algorithm assigns 0 to the left link of every node and 1 to its right link. The code for a leaf/symbol is a concatenation of link values from the root to the leaf. For example, assume an alphabet consists of 6 elements of A, B, C, D, E and F with probabilities of occurrences of 0.4, 0.2, 0.14, 0.11, 0.1 and 0.05, respectively. The binary tree will be formed as in Fig. 8 in the appendix.

In the above example, the father node "0.15" of symbols E and F was formed because their P(E) and P(F) are the smallest, then the father node "0.25" of symbols C and D was formed, then the father node "0.35" of symbol B and node "0.15", and then the father node "0.6" of nodes "0.35" and "0.25". Finally, the root node "1.0" from his two sons of node "0.6" and symbol A was formed. The coding table of compression is formed as follows:

Input symbols	Output codes
A	1
B	000
C	010
D	011
E	0010
F	0011

Table 8 Huffman coding for the example in Fig. 8 in the appendix.

Groups III and IV facsimile (fax) are using some sort of Huffman encoding for data compression [CCITT, Hunte] (earlier groups such as Groups I and II do not use any data compression scheme and they are slow). For examples, the black run length (number of black pels) of 3 is coded in short bits 10 and the black run length of 8 is coded in long bits 000101 because the former occurs more frequently than the latter.

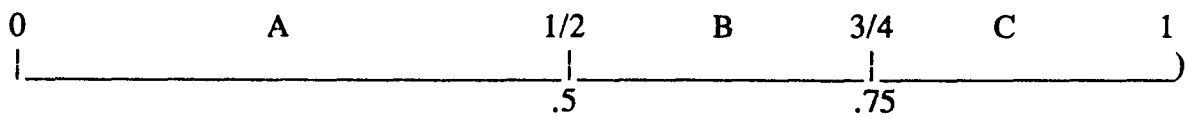
3.2.2 Arithmetic Coding

Arithmetic coding uses a different approach. It uses a real number between 0 and 1 to encode an entire input message file. The algorithm is described as follows: The interval is initially [0,1), i.e., the interval between 0 and 1, and 0 is included but 1 is excluded. It is partitioned according to the probabilities of the alphabet used. A symbol of the input message is coded with the subinterval corresponding to that symbol. At the next iteration the subinterval will also be narrowed: the subinterval is partitioned proportionally according to the probabilities of the alphabet, and a new subinterval is chosen in the same way. The process then continues until all input symbols are read and processed. The newest subinterval or any number inside that interval is the compressed

result. The expansion algorithm decodes an interval or a number into its original character string in a similar way.

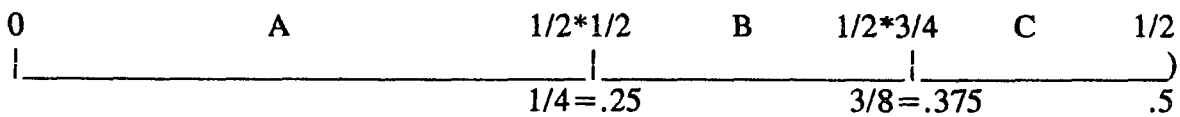
For example, an alphabet consists of three symbols: A, B and C with probabilities of occurrences 1/2, 1/4 and 1/4, respectively. Assume an input data file is ABCB. The encoding of the string ABCB is as follows:

Initially, the interval is [0, 1). After the first character A is read, the interval is narrowed to [0, 1/2) as shown in Fig. 9, each subsequent character that is read will also narrow the intervals as shown in figures 10, 11 and 12, respectively.



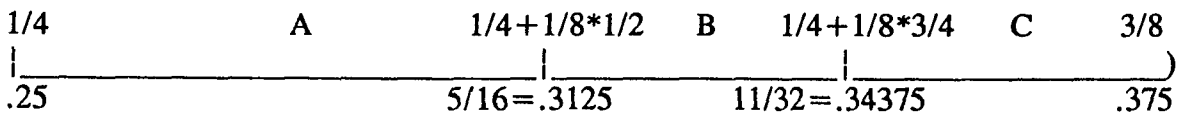
Interval width is 1 initially.

Fig. 9 Arithmetic coding after the first character A is read, the interval [0, 1) is narrowed to [0, 1/2) with the new width 1/2.



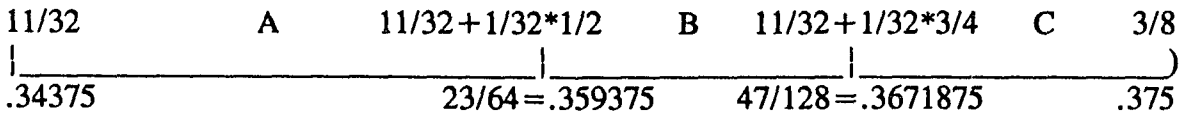
Interval width is now = length of [0, 1/2) = 1/2.

Fig. 10 Arithmetic coding after the second character B is read, the interval [0, 1/2) is narrowed to [1/4,3/8) with the new width 1/8. Note, the interval is zoomed in for readability, the same is true for the following two figures, it is just like using more powerful magnifying glasses to inspect this figure and each of the following two figures.



Interval width is now = length of [1/4, 3/8) = 1/8.

Fig. 11 Arithmetic coding after the third character C is read, the interval [1/4, 3/8) is narrowed to [11/32,3/8) with the new width 1/32.



Interval width is now = length of [11/32, 3/8) = 1/32.

Fig. 12 Arithmetic coding after the fourth character B is read, the interval [11/32, 3/8) is narrowed to [23/64, 47/128) with the new width 1/128.

Therefore the output code of the input string ABCB is any real number in the interval [23/64, 47/128) or [0.359375, 0.3671875). For example, it can be 23/64, i.e., 0.359375 or 0.36 or just the interval [23/64, 47/128). 0.36 is chosen to be the compressed result to illustrate the decoding process.

The decoding of the encoded value of 0.36 is as follows:

When the first digit 3 is read or received, we know that we can have the number in the interval [0.3, 0.4). Since the interval is within the first half subinterval of [1, 0), we know that the first symbol must be A and the first subinterval must be [0, 1/2). Then we can partition the interval [0, 1/2) as in Fig. 10. The interval [0.3, 0.4) is not totally within any subinterval in Fig. 10, hence we need to read another digit. In this case we read the second digit to be 6, so we now have the new interval [0.36, 0.37). We check which second symbol will have the subinterval which will include the above interval. By inspection, if the second symbol is A we will have subinterval [0, 1/4) which does not contain the interval, if it is B we will have the subinterval [1/4, 3/8) = [0.25, 0.375) which totally contains the interval [0.36, 0.37) as shown in Fig. 10. Therefore, we find the second character to be B. Then we can partition the interval [1/4, 3/8) as shown in Fig. 11. The interval [0.36,0.37) is totally within the subinterval [11/32, 3/8), hence the third character must be C. Now, we partition the interval [11/32, 3/8) as shown in Fig.

12. We want to see if the interval [0.36, 0.37) is within any subinterval in Fig. 12. In this case, we cannot find any. Hence we need to read another digit. But we don't have any more digits. Since 0.36 can also be interpreted as 0.360, i.e., we have the interval [0.360, 0.361), which interval is entirely within the subinterval [23/64, 47/128). Therefore, our fourth symbol must be B. Since 0.360 can also be interpreted as 0.3600, 0.36000 and so on, the process of expansion can be continued until we find the entire input file. The process of expansion needs either to know the length of the input file or to encounter a terminating symbol in order to stop after the last character of the input file is decoded.

3.2.3 Lempel-Ziv (LZ) Coding

The Lempel-Ziv (LZ) algorithm uses the method of substitution. The algorithm uses a pointer and a length to point back to a previously encountered substring which is the longest.

For example, assume an input data file is

```
ababaabaabaabaaabbbbbaa
 1 3 5 7 9 1 3 5 7 9 1 3
```

These number are the positions of the symbols, only some odd number positions are shown.

The encoding of the above input will be

```
a b (1,3) (3,10) (12,2) (17,4) (15,2)
```

where the first number in a bracket is the position of the first symbol of the longest matching string, and the second number in a bracket is the length of the matching strings.

Ziv and Lempel have two implementations, called LZ77 and LZ78. Lempel-Ziv and Welch (LZW) algorithm is a *modified* version of LZ78, and is widely used in the industry. These three algorithms will be briefly described here:

3.2.3.1 LZ77 Algorithm

LZ77 uses a triplet to include a pointer to a previous encountered string which is the longest. Every triplet consists of the following:

<i, j, a>

where **i** is the offset from the current string to the previously encountered longest string, **j** is the length of the matching strings, and **a** is the first character after the current matching string. LZ77 uses a sliding window so that it limits how far the algorithm will go back, and it also has a limit for the length of the matching strings.

For example, an input data file is the same as before:

```
ABABAABAABAABAAABBBBBAA
 1 3 5 7 9 1 3 5 7 9 1 3
```

The encoding of the above string will be

```
(0,0,A) (0,0,B) (2,3,A) (3,9,A) (4,1,B) (1,3,A) (0,0,A)
```

3.2.3.2 LZ78 Algorithm

The LZ78 algorithm parses an input string into groups, each group consists of a

matching string and a non-matching character. Each group is numbered. The encoding consists of a series of pairs of a pointer and a character. The pointer points to the longest matching string, and the character is a non-matching character.

For example, an input data file is the same as before:

```
ABABAABAABAABAAABBBBBAA
 1 3 5 7 9 1 3 5 7 9 1 3
```

We can group and number the above input as follow:

```
A  B  AB  AA  BA  ABA  ABAA  ABB  BB  BAA
 1  2  3   4   5    6     7    8    9   10
```

The compressed result will be as follows:

```
(0,A)(0,B)(1,B)(1,A)(2,A)(3,A)(6,A)(3,B)(2,B)(5,A)
```

The above result can also be represented as a trie data structure as shown in Fig. 13 in the appendix.

3.2.3.3 Lempel-Ziv and Welch (LZW) Algorithm

An improved variation of LZ78 algorithm, the Lempel-Ziv and Welch (LZW) algorithm [Welch] is briefly described here. LZW has been widely implemented commercially due to its fast speed, simplicity and competitive compression ratio. The compressor builds a string table which contains strings which occur in an input data file. Assume all single character strings are also in the string table. Suppose a string α is read

from an input data file and is matched with a string in the string table (initially, α is a single character), an extension character K is then read from the data file, LZW searches the string table to see whether there is a string αK: if there is none, then the position of α in the string table is outputed as an output code and the string αK is put in the string table (this is called the **"greedy method"**), else make αK to be the new α and a new character K is read from the data file and the above process is repeated until the end of the input data file is reached. LZW decompressor mimics the process of the compressor in forming the string table and outputing a character string from each code of the compressed file by consulting and updating the string table. An effective method of building the string table and then accessing it can be found in [McCre].

LZW uses a dictionary of size of $2^{12} = 4096$. The compressed result of the LZW algorithm is a series of pointers of size 12 bits.

For example, The input data file is the same as before:

ABABAABAABAABAAABBBBBAA

1 3 5 7 9 1 3 5 7 9 1 3

After the above input data file has been processed, the string table will be shown as in the following string table:

String Table

A	1
B	2
AB	3
BA	4
ABA	5
AA	6
ABAA	7
ABAAB	8
BAA	9
AAB	10
BB	11
BBB	12
BBA	13

The compressed result will be as follows:

1 2 3 1 5 7 4 6 2 11 11 6

3.2.4 Proposed Solution

The thesis is to describe a new lossless data compression/expansion algorithm which overcome some of the shortcomings of the LZW algorithm [Chang91B]. It competes favorably with other compression algorithms [Copel, Corte, Fiala, Pechu, Tropp] in balance of performance, speed and working storage requirement. For a survey

of data compression see Ref. [Lelew].

The algorithm uses multiple (two will be used as an example) dictionaries for storing previously encountered strings of a data file to be compressed. The first dictionary is a short one, it has 128 entries and needs 7 bits to store its address. The second dictionary is a long one, it has 32,768 entries and requires 15 bits to store its address. Each of these two dictionaries is initialized with frequently occurring strings, and organized as a binary tree. The short dictionary of course contains still more frequently occurring strings. Each of these dictionaries is organized in a **lexicographic** tree. An input data file is compared with the dictionaries so that the longest substring which can be found in a dictionary will give its position in the dictionary as an output code. By using the "greedy method", the algorithm stores new character strings in the dictionaries for future references. Compression and expansion use the same procedure to form and update dictionaries. A compressed result consists of a series of pointers to the dictionaries with a byte as a pointer to the short dictionary and two bytes as a pointer to the long dictionary. During compression/expansion, the more frequently occurring strings will be swapped into the short dictionary. The two dictionaries are used locally as scratch pads and need not be stored/transmitted.

3.2.5 Design Consideration of the New Algorithm

The objective here is to design a universal compression algorithm such that it can compress a wide variety of data files without prior knowledge of their statistics. An **adaptive** scheme is preferred since a good non-adaptive scheme often requires two passes

to process the input data: in the first pass it gathers statistics and in the second pass it performs the compression job. A good adaptive scheme executes at most only marginally worse than a two-pass non-adaptive scheme [Bell], most often the former is much better than that of the latter because it does not need to store or transmit the statistics. Of course, the speed of the former is much better than that of the latter. For the compression of English text files, Huffman and arithmetic codings need to form an enormous table for even 4-character strings because they need 4,294,967,296 bytes just to store these statistics (each byte consists of 8 bits, and all combination of 8 bits is 2^8 = 256, then 256^4 = 4,294,967,296). LZW can remember long strings in its string table of reasonable size, the string table just contains previously occurring strings, not just a combination of an alphabet. LZW is much widely used as a universal data compression than Huffman and arithmetic codings because of its fast speed and competitive performance compared with the others. For example, two popular compression software ARC and PKzip use LZW algorithm, also does the AT&T compress routine.

The algorithm which is presented in the thesis is a modified version of LZW and corrects LZW's major shortcomings. It will be called **LZWC** algorithm in the following discussion.

3.2.6 Advantages of the Proposed New LZWC Algorithm

LZW has the following shortcomings:

1. In LZW, if a string is in the string table, then all its proper prefix substrings are also in the string table. Therefore, a potential problem with LZW is the overflow of its

string table. If a string table is overflowed, the string table has to be flushed, and the efficiency is lost.

2. In LZW, initially the dictionary contains only single characters, therefore the compressed result is actually an expansion of 12/8 - 1 = 50%. Only after a reasonable number of strings have been built into the dictionary, then the result shows the advantages of the algorithm.

3. In LZW, the pointer size is fixed at 12 bits. Can we do better?

The new proposed LZWC algorithm address the above major problems of LZW.

1. LZWC does not include all proper prefixes of a string which is in a dictionary to save storage space of the dictionaries. It links dictionary entries in lexicographic order. Therefore, any entry in a dictionary can be deleted so that its place can be used for a new entry. LZWC needs never be refreshed like LZW because LZWC will never overflow, since the least recently used entry will just be swapped out. It can handle a very large data file without needing to reset the string tables.

2. LZWC uses two dictionaries (from now on they are also called trees because each dictionary is organized in a binary tree form, also called string tables because they are mainly two tables that store strings) instead of one used in LZW. Because the total number of entries of the two dictionaries is $2^7 + 2^{15} = 128 + 32768 = 32896$, a simple English dictionary can be contained in these two tables. Therefore, compression result is good for English texts even at the beginning of the compression.

3. LZWC uses two dictionaries, the most frequently occurring strings use only 8 bits for pointers to the short dictionary.

3.2.7 LZWC Algorithm

The LZWC algorithm is an adaptive data compression/expansion scheme, which uses two dictionaries for storing strings. A short dictionary has 128 entries and requires an address of 7 bits to point to any of its entries. A long dictionary has 32,768 entries and needs an address of 15 bits to point to any of its entries. These two dictionaries are initialized with entries of all 256 permutations of 8 bits (because of the popularity of 8/16/32 bits computers including PCs) and other frequently occurring strings based on the statistics of the English language and graphical images. The short dictionary, of course, contains still more frequently occurring strings. Each of these dictionaries is organized in a binary tree of lexicographic order. An input data file to be compressed is read character by character such that the longest substring α which matches an entry in one of the two dictionary trees is obtained, and the position of the dictionary is outputed, for the short dictionary one bit of value zero is appended to the front of the other 7 bits, for the long dictionary one bit of value 1 is appended to the front of the other 15 bits. Therefore, the output codes pointing to the short and long dictionaries are always one byte or two bytes, respectively. The frequency count of the string α in the dictionary is incremented every time α is referenced. If the string α is in the long dictionary and its count is greater than any in the short dictionary, then the swapping of these two entries will be performed. A next character K is read from the input file, and the string αK will be put into the long dictionary (this is again the "greedy method"). If the long dictionary is already full, the least recently used entry will be replaced by the new entry αK. Now, we make K to be the new α. The process of compression continues until the end of the

input data file is reached. The expander (also known as decompressor) mimics the process of the compressor in forming and updating the dictionaries and outputs a character string from each code read from the compressed data file. The two dictionaries are thus used as local scratch pads during the algorithm execution and need not be stored/transmitted otherwise.

LZWC updates the two dictionaries during the process of compression/expansion. The compressor and the expander use the same initial dictionaries and the equivalent algorithms to update dictionaries so that the dictionaries will be synchronized. For readability, the LZWC algorithm of compression/expansion is described below in a pseudo programming language.

There is a special case which we wish to consider. It is described here as an improvement of the LZWC algorithm for a particular case. Assume the input string is KμKμK where K is a character and μ is a string. Suppose the string Kμ is already an entry x in a dictionary. When we see the second "K", suppose "KμK" is not in any of the dictionaries, by the "greedy method" the compressor puts "KμK" in the long dictionary, say entry y. Then the substring KμK starting from the second "K" would be encoded as y. Therefore, the whole string KμKμK is encoded as xy. When the expander receives the code xy, from x, the expander knows that the string corresponding to x is Kμ; when the expander sees a y, the entry y in the dictionary has not been created by the expander yet because the first character "K" of the string corresponding to code y has not been decoded yet. But we know in this case that the code y would point beyond the current entries if the long dictionary is not full yet, and the string corresponding to y

must be KμK where Kμ is the previous substring which has been decoded. Therefore the expander will put KμK into the long dictionary as y entry of the dictionary. When the dictionaries are already full and the compressor sees the string "KμKμK" as described above, the compressor will encode the string "KμKμK" as xx for the substring "KμKμ" and then put "KμK" in the long dictionary by replacing the least recently used entry. The expander also follows this procedure.

3.2.7.1 The Pseudo Code of the LZWC Compression

Initialize the two dictionaries with single-character strings, other more frequently occurring strings for English text and some strings for representing graphical images by using a statistics data file. Each dictionary is in a binary tree form.

α <- '\0'. /* α is the current substring of the input file, it is initialized with an empty string '\0' */

μ. /* μ is a substring of α and μ can be found in either the short or the long dictionaries */

flag <- 0. /* A flag for putting a string in the long dictionary in the next run by the greedy method */

ß. /* A buffer for storing a string by greedy method */

While (the end-of-file of the input file not reached) do

{

Read next character K of the input data file.

While (αK is found to be an entry in a dictionary) do

{

μ <- αK.

α <- αK.

If K is empty, then break, i.e., exit the innermost while loop.

Read next character K from the input data file.

If K is empty, then break.

}

saved_code <- position of μ in the dictionary.

If the dictionary is the short one, set the 8th bit (the rightmost bit is the 1st bit, and counting is from right to left) of saved_code to be 0, and output its lower byte;

else set its 16th bit to 1, and output the two bytes of saved_code.

If K is empty then compression is finished and we are done.

Increment the count of occurrences of the entry μ.

If (flag is equal to 1)

{

/* Use the string ß saved by the greedy method in the last run */

If ß is not in any dictionary, put the string ß in the long dictionary by replacing the least recently used string in that dictionary.

}

/* Now, use the greedy method to try to put the string μK into the long dictionary. */

If the long dictionary is not full, then

{

Put the string μK into the next available slot of the long dictionary.

}

else

{

/* The long dictionary is full, we set flag so that we wait until the next cycle time to put the greedy string in the long dictionary as explained above for the case of KμKμK */

ß <- μK. /* save the string for the use in the next cycle */

flag <- 1. /* Set flag, so that we can do next time */

}

α <- K. /* start again with the new α */

}

3.2.7.2 The Pseudo Code of the LZWC Expansion

Initialize the two dictionaries with single-character strings, other more frequently occurring strings and some strings for representing graphical images in the same way as the compressor inserts strings in the dictionaries by using the same statistics data file. Each dictionary is in a binary tree form, same as in compression.

ß <- '\0'. /* A buffer for storing a string by greedy method */

Read code **i** of one byte of an input compressed file.

While the code **i** is not empty do

{

If the most significant bit of the code **i** is 0

{

From the **i**th entry of the short dictionary

{

Find the string μ and output it.

Increment the count of μ in the dictionary.

}

}

else

{

/* It is the long dictionary */

Read next byte **j**.

Strip the most significant bit of the code **i**.

m <- **i** * 256 + **j**.

If the long dictionary is full or **m** is not out of bound of the current entries, then

{

From the **m**th entry of the long dictionary find the string μ and output it.

Increment the count of μ in the dictionary.

If μ's count is greater than that of an entry in the short dictionary,

```
                then swap these two entries.
            }
        else
            {
            /* Kμ K case */
            /* ß is the previously saved substring, and F denotes the first
            character of ß */
            Output the string ßF.
            μ <- ßF.
            }
        }
    /* K denotes the first character of μ */
    If ßK is not in any dictionary, put ßK in the long dictionary either in the empty
    slot or by replacing the least recently used entry.
    ß <- μ.
    Read code i of the input compressed file.
    }
```

3.2.8 A Statistics Data File for Building Initial Dictionaries

A statistics data file of strings with high frequency of occurrences is created according to their descending order of the frequency of occurrences. The file will include all 8-bits combinations, most frequently occurring English words [Copel], and some run

lengths of null characters and hex F characters for images. The statistics data file will be used to form the initial dictionaries in lexicographic order. It is obvious that the same statistics data file must be used by both compressor and expander to build the same initial dictionaries.

Assume the part of the statistics data file that builds the long dictionary is shown as follow:

the be of and in he to have it for they with not then on she as at by this we you from do ...

The tree structure of a part of the long dictionary is shown in Fig. 14 in the appendix after the long dictionary is constructed.

Knuth [Knuth] has an algorithm of forming an optimum **binary search tree** (binary tree with lexicographic order) if the statistics of the entries are known.

3.2.9 Dictionaries

Since LZWC dictionaries are constantly changing during compression or decompression, no attempt is made to build optimum binary search trees in the LZWC algorithm. However, initially the entries to be put into the dictionaries can be arranged in non-increasing order of frequency. In doing a particular example of [Knuth], by forming a binary search tree this way (see Fig. 15), the method also happened to be the optimum one.

There are four string operations to be considered: matching, inserting, replacing and swapping.

3.2.9.1 Matching

Initially, a variable "node_ptr" points to the root. A string to be matched with a dictionary is compared with the node_ptr. If the string is smaller than the string in the node_ptr in lexicographic order, then the left son of the node_ptr is chosen, else the right son of the node_ptr is chosen, and it is the new node_ptr. Then compare the string with the new node_ptr. The process continues until we find a matching node or report a failure.

3.2.9.2 Inserting

Use the method of matching strings described above. The node to be inserted is always to be a leaf of the tree.

3.2.9.3 Replacing

Replacement consists of a deletion and an insertion. The deletion algorithm is as follows:

Since all nodes in each dictionary are also linked together by the value of their count, we also use two pointers to point to the head and tail of each chain. The tail node of the chain which has the smallest count of occurrence is a candidate for deletion. The place of the node that is to be deleted will be used for storing a new entry, but the link between the node and his father will not be used, also the links between it and its two sons will not be used. The new entry in the node will be inserted into the tree.

The LZWC algorithm will release and allocate memory storage for storing

storing strings.

3.2.9.4 Swapping

Use the method of replacing strings, we can easily find a candidate in the short dictionary for replacement. Swap the strings, release and allocate memory storage for storing these two strings.

3.2.10 Error Susceptibility

Because output codes of the LZWC algorithm are either a byte or two bytes, one type of bit error, amplitude error [Lelew] will not propagate easily if the most significant bit of a byte/word pointer is not corrupted.

3.2.11 Example

The two dictionaries can be initialized with single-characters and frequently occurring strings. An example of a short dictionary and a part of a long dictionary are shown in Table 9. (For easy readability, whole strings are placed inside the dictionaries, however, the algorithm requires that its first character is placed in a dictionary and the remaining characters are dynamically stored somewhere else).

Short Dictionary				Long Dictionary			
0	e	64	z	0	Q	64	than
1	t	65	3	1	X	65	them
2	a	66	6	2	Z	66	can
3	(sp)	67	4	3	^	67	only
4	o	68	7	4	up	68	other
5	n	69	8	5	my	69	new

6	i	70	V	6	me	70	some
7	r	71	Y	7	us	71	could
8	s	72	$	8	go	72	time
9	h	73	:	9	am	73	these
10	l	74	_	10	st	74	two
11	d	75	~	11	de	75	may
12	c	76	(	12	do	76	then
13	u	77	)	13	du	77	first
14	m	78	*	14	's	78	any
15	f	79	;	15	bi	79	now
16	(Tab)	80	{	16	re	80	such
17	(CR)	81	}	17	il	81	like
18	(LF)	82	the	18	op	82	our
19	p	83	of	19	the(sp)	83	over
20	g	84	and	20	ir	84	man
21	y	85	to	21	ment	85	even
22	w	86	in	22	est	86	most
23	b	87	is	23	ments	87	made
24	,	88	ed	24	ions	88	after
25	,(sp)	89	er	25	that	89	also
26	.	90	not	26	was	90	did
27	.(sp)	91	s(sp)	27	for	91	many
28	v	92	d(sp)	28	with	92	before
29	k	93	es	29	his	93	must
30	T	94	ly	30	this	94	through
31	S	95	ing	31	had	95	back
32	&	96	ion	32	ers	96	years
33	A	97	un	33	are	97	where
34	M	98	a(sp)	34	but	98	much
35	C	99	he	35	from	99	your
36	"	100	The	36	have	100	way
37	-	101	it	37	they	101	well
38	H	102	as	38	which	102	00
39	I	103	on	39	one	103	000
40	B	104	be	40	you	104	0000
41	E	105	at	41	were	105	00000
42	'	106	by	42	her	106	(sp)(sp)
43	0	107	or	43	all	107	(sp)(sp)(sp)
44	1	108	an	44	she	108	(FF)(FF)
45	P	109	we	45	there	109	(FF)(FF)(FF)
46	R	110	no	46	and(sp)	110	(FF)(FF)(FF)(FF)
47	N	111	if	47	would	111	(00)
48	D	112	so	48	their	112	(00)(00)
49	L	113	(CR)(LF)	49	him	113	(00)(00)(00)

50	x	114	!	50	been	114	(00)(00)(00)(00)
51	O	115	%	51	has	115	(01)
52	W	116	+	52	when	116	(02)
53	F	117	/	53	who	117	(03)
54	G	118	<	54	will	118	(04)
55	#	119	=	55	more	119	(05)
56	J	120	>	56	ings	120	(06)
57	2	121	?	57	ments(sp)	121	(07)
58	5	122	@	58	out	122	(08)
59	K	123	[	59	said	123	(09)
60	U	124	\	60	what	124	(0B)
61	j	125	]	61	its	125	(0C)
62	9	126	¦	62	about	126	(0E)
63	q	127	‘	63	into	127	(0F)
						128	(10)
						129	(11)
						.	.
						.	.
						.	.
						32767	.

Table 9 An example of dictionaries where
(sp) represents a space character. (CR)= Carriage Return
(LF)= Line Feed. (FF) is a byte with all its 8 bits set to 1.
(00) is a byte with all its 8 bits set to 0.
(01)=a byte with hex value of 01, (02)=a byte with hex value of 02, and so on.

To demonstrate the effectiveness of LZWC algorithm, the following sample English text file in Table 10 is to be compressed:

Four score and seven years ago, our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we can not dedicate-we can not consecrate-we can not hallow this ground. The brave men, living and dead who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us-that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion-that we here highly resolve that these dead shall not have died in vain-that this nation, under God, shall have a new birth of freedom-and that government of the people, by the people, for the people shall not perish from the earth.

Table 10 Abraham Lincoln's Address at Gettysburg, Pennsylvania.

If we assume a tab before every paragraph, a single space between adjacent words on a same line and a CR and a LF after each line, the original text occupies 1473 bytes storage. Since the two dictionaries can have 32K words, we can reasonably assume almost all English words in Table 10 can be found from one of the dictionaries. The final result is that the compressed file needs 310 one-byte pointers and 161 two-byte pointers. Also we find that the number of distinct 1-byte pointers is less than 128, the size of the short dictionary. Therefore, the total storage required is 310 + 161 * 2 = 632 bytes. The compression ratio = 1473/632 = 2.33, and the compression percentage = 632/1473 = 42.9%.

A popular shareware compression program PKZIP/PKUNZIP [Vaugh] version

0.92, which uses the LZW compression algorithm, was used to experiment the aforementioned example file. The compressed result has 978 bytes. Among these 978 bytes, 104 bytes are used for header information, only the remaining 978 - 104 = 874 bytes are for the file itself. Hence, the compression ratio = 1473 / 874 = 1.69, and the compression percentage = 874 /1473 = 59.3%.

By comparing the results from PKZIP and LZWC, we can certainly see that the result of LZWC is substantially better than that of the PKZIP program.

3.3 Controlled Redundancy

The aim of data compression is to try to reduce as much redundancy of a data file as possible; on the other hand, a damaged file which has been compressed very effectively can hardly be deciphered, if not impossible. Therefore, some controlled redundancy can be purposely put into the compressed file such that we can recover the original file from its damaged compressed counterpart if the damage does not exceed a certain threshold. One such method was suggested by Rabin [Rabin], and was called **Information Dispersal Algorithm (IDA)**. IDA can split a file F of length L into L/m pieces, each of length m. By manipulating these pieces, n (> m) messages can be formed, any m messages among these n can be used to reconstruct the original file. For example, suppose we assume n = 17 and m = 16, a file F of length L=64000 can be splitted and manipulated into n=17 pieces, each of which is of length L/16 = 64000/16 = 4000. If any one piece is lost or damaged the other 16 pieces can be used to easily reconstruct the original file. In this case, the amount of redundancy is about (17-16)/16

= 6.25%. A simple illustration of IDA can be seen from Ref. [Chang] or the beginning part of the following discussion.

Now, some notations. IDA first breaks a file F of length L into pieces of length L/m. By manipulating these pieces, n subfiles F_i (the length of F_i, $|F_i|$ is equal to L/m), where $1 \leq i \leq n > m$, are constructed. Every m messages of F_i's suffice for reconstructing the original file F. Since n/m can be chosen to be a little bit over 1, IDA is space efficient. IDA can be used to transmit a file reliably even in case of loss of n-m messages due to failure of nodes and/or links. It can also be used to reliably store files on a single disk. IDA can be made time efficient by choosing suitable m, n and vectors $\mathbf{a}_i$'s, $1 \leq i \leq n$, where $\mathbf{a}_i$'s are used to manipulate pieces of the original file F.

We will explain how to design m, n and vectors $\mathbf{a}_i$'s so that the original file F can be easily splitted and manipulated into n pieces and later any m pieces from these n pieces ($n > m$) can be used for reconstructing the original file F efficiently.

3.3.1 Splitting Files

To illustrate IDA, an example of file F of length L=16 is used. We denote that F consists of 16 characters b_i's (each character can be considered as an 8-bit unsigned integer)[††], and we segment F into sequences of length L/m=16/2=8. Thus

$F = (b_1,b_2)(b_3,b_4)(b_5,b_6)(b_7,b_8)(b_9,b_{10})(b_{11},b_{12})(b_{13},b_{14})(b_{15},b_{16})$

[††]Because of the popularity of 8/16/32-bit computers, we consider each character as an 8-bit unsigned integer. In the thesis, we normally use modulo $256=2^8$ arithmetic. As we will see later, sometimes we need more than 8 bits to store a character. In this case we will use other than 256 for modulo arithmetic. For example, if 9 bits are used to store a character, then $2^9=512$ will be used for modulo arithmetic.

We choose n=4 pairwise linearly independent vectors:

$\mathbf{a}_1 = (1, 0)$

$\mathbf{a}_2 = (0, 1)$

$\mathbf{a}_3 = (1, 1)$

$\mathbf{a}_4 = (1, 3)$

We can construct n=4 subfiles, and each subfile is of length $| F_i | = L/m = 16/2 \cdot 8$:

(Of course we need a byte or some small place to store the number i in F_i. For the simplicity of illustration of the examples we do not include i in F_i.)

Denote F_i's by the following character strings:

$F_1 \equiv c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{17}, c_{18}$

$F_2 \equiv c_{21}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{27}, c_{28}$

$F_3 \equiv c_{31}, c_{32}, c_{33}, c_{34}, c_{35}, c_{36}, c_{37}, c_{38}$

$F_4 \equiv c_{41}, c_{42}, c_{43}, c_{44}, c_{45}, c_{46}, c_{47}, c_{48}$

where

$$c_{11} = \mathbf{a}_1 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = (1\ 0) \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b_1$$

$$c_{12} = \mathbf{a}_1 \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = (1\ 0) \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = b_3$$

$$c_{13} = \mathbf{a}_1 \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = (1\ 0) \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = b_5$$

$$c_{14} = \mathbf{a}_1 \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = (1\ 0) \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = b_7$$

$$c_{15} = \mathbf{a}_1 \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = (1\ 0) \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = b_9$$

$$c_{16} = a_1 \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = (1\ 0) \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = b_{11}$$

$$c_{17} = a_1 \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = (1\ 0) \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = b_{13}$$

$$c_{18} = a_1 \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = (1\ 0) \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = b_{15}$$

Therefore, we have

$$F_1 = b_1, b_3, b_5, b_7, b_9, b_{11}, b_{13}, b_{15} \tag{3.1}$$

Similarly,

$$c_{21} = a_2 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = (0\ 1) \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b_2$$

$$c_{22} = a_2 \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = (0\ 1) \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = b_4$$

$$c_{23} = a_2 \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = (0\ 1) \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = b_6$$

$$c_{24} = a_2 \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = (0\ 1) \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = b_8$$

$$c_{25} = a_2 \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = (0\ 1) \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = b_{10}$$

$$c_{26} = a_2 \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = (0\ 1) \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = b_{12}$$

$$c_{27} = a_2 \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = (0\ 1) \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = b_{14}$$

$$c_{28} = a_2 \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = (0\ 1) \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = b_{16}$$

Therefore, we have

$$F_2 = b_2, b_4, b_6, b_8, b_{10}, b_{12}, b_{14}, b_{16} \tag{3.2}$$

Also,

$$c_{31} = a_3 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = (1\ 1) \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b_1 + b_2$$

$$c_{32} = a_3 \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = (1\ 1) \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = b_3 + b_4$$

$$c_{33} = a_3 \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = (1\ 1) \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = b_5 + b_6$$

$$c_{34} = a_3 \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = (1\ 1) \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = b_7 + b_8$$

$$c_{35} = a_3 \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = (1\ 1) \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = b_9 + b_{10}$$

$$c_{36} = a_3 \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = (1\ 1) \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = b_{11} + b_{12}$$

$$c_{37} = a_3 \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = (1\ 1) \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = b_{13} + b_{14}$$

$$c_{38} = a_3 \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = (1\ 1) \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = b_{15} + b_{16}$$

Therefore, we have

$$F_3 = b_1+b_2,\ b_3+b_4,\ b_5+b_6,\ b_7+b_8,\ b_9+b_{10},\ b_{11}+b_{12},\ b_{13}+b_{14},\ b_{15}+b_{16} \qquad (3.3)$$

Also,

$$c_{41} = a_4 \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = (1\ 3) \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = b_1 + 3 * b_2$$

$$c_{42} = a_4 \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = (1\ 3) \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} = b_3 + 3 * b_4$$

$$c_{43} = a_4 \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = (1\ 3) \begin{bmatrix} b_5 \\ b_6 \end{bmatrix} = b_5 + 3 * b_6$$

$$c_{44} = a_4 \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = (1\ 3) \begin{bmatrix} b_7 \\ b_8 \end{bmatrix} = b_7 + 3 * b_8$$

$$c_{45} = a_4 \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = (1\ 3) \begin{bmatrix} b_9 \\ b_{10} \end{bmatrix} = b_9 + 3 * b_{10}$$

$$c_{46} = \mathbf{a}_4 \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = (1\ 3) \begin{bmatrix} b_{11} \\ b_{12} \end{bmatrix} = b_{11} + 3 * b_{12}$$

$$c_{47} = \mathbf{a}_4 \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = (1\ 3) \begin{bmatrix} b_{13} \\ b_{14} \end{bmatrix} = b_{13} + 3 * b_{14}$$

$$c_{48} = \mathbf{a}_4 \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = (1\ 3) \begin{bmatrix} b_{15} \\ b_{16} \end{bmatrix} = b_{15} + 3 * b_{16}$$

Therefore, we have

$$F_4 = b_1 + 3 * b_2,\ b_3 + 3 * b_4,\ b_5 + 3 * b_6,\ b_7 + 3 * b_8,\ b_9 + 3 * b_{10},\ b_{11} + 3 * b_{12},\ b_{13} + 3 * b_{14},\ b_{15} + 3 * b_{16} \qquad (3.4)$$

We can transmit or store these four F_1, F_2, F_3 and F_4 instead of the original file F.

3.3.2 Recombining Files

3.3.2.1 Suppose we have only two subfiles F_2 and F_3 (out of the total four subfiles F_1, F_2, F_3 and F_4), we then form a matrix A_{32} with a_3 and a_2 as its rows:[†††]

$$A_{32} = \begin{bmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

(note that we use A_{32} instead of A_{23}, and arrange A_{32} as

$\begin{bmatrix} \mathbf{a}_3 \\ \mathbf{a}_2 \end{bmatrix}$, not as $\begin{bmatrix} \mathbf{a}_2 \\ \mathbf{a}_3 \end{bmatrix}$)

$$A_{32} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}^{-1}$$

Then by the definition of c_{21} and c_{31} we have:

$$A_{32} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_{31} \\ c_{21} \end{bmatrix}$$

[†††]The first two vectors are put into their original positions if they exist. Now, because the message corresponding to a_1 is missing, we put a_3 in place of a_1. This way, it will make the Gauss elimination method to find the inverse of a matrix very easy.

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = A_{32}^{-1} \begin{bmatrix} c_{31} \\ c_{21} \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_{31} \\ c_{21} \end{bmatrix} = \begin{bmatrix} c_{31} - c_{21} \\ c_{21} \end{bmatrix} \qquad (3.5)$$

Therefore,

$b_1 = c_{31} - c_{21}$

$b_2 = c_{21}$

Similarly, we can derive other b_i's. From (3.2) and (3.3) we have

$F_2 \equiv c_{21}, c_{22}, c_{23}, c_{24}, c_{25}, c_{26}, c_{27}, c_{28}$

$= b_2, b_4, b_6, b_8, b_{10}, b_{12}, b_{14}, b_{16}$

$F_3 \equiv c_{31}, c_{32}, c_{33}, c_{34}, c_{35}, c_{36}, c_{37}, c_{38}$

$= b_1+b_2, b_3+b_4, b_5+b_6, b_7+b_8, b_9+b_{10}, b_{11}+b_{12}, b_{13}+b_{14}, b_{15}+b_{16}$

Finally, by using the result in (3.5) we form a character string from F_2 and F_3 as follows:

$c_{31} - c_{21}, c_{21}, c_{32} - c_{22}, c_{22}, c_{33} - c_{23}, c_{23}, c_{34} - c_{24}, c_{24},$

$c_{35} - c_{25}, c_{25}, c_{36} - c_{26}, c_{26}, c_{37} - c_{27}, c_{27}, c_{38} - c_{28}, c_{28}$

$= b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, b_{15}, b_{16}$

$=$ F (the original file)

3.3.2.2 Suppose now we have only two subfiles F_1 and F_4 (out of the total four subfiles F_1, F_2, F_3 and F_4), we then form a matrix A_{14}:

$$A_{14} = \begin{bmatrix} \mathbf{a_1} \\ \mathbf{a_4} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 3 \end{bmatrix}$$

$$A_{14}^{-1} = \begin{bmatrix} 1 & 0 \\ 1 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 \\ -1/3 & 1/3 \end{bmatrix}$$

Then by the definition of c_{11} and c_{41} we have:

$$A_{14} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_{11} \\ c_{41} \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = A_{14}^{-1} \begin{bmatrix} c_{11} \\ c_{41} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -1/3 & 1/3 \end{bmatrix} \begin{bmatrix} c_{11} \\ c_{41} \end{bmatrix} = \begin{bmatrix} c_{11} \\ (-c_{11} + c_{41})/3 \end{bmatrix} \qquad (3.6)$$

Therefore,

$$b_1 = c_{11}$$

$$b_2 = (c_{41} - c_{11}) / 3$$

Similarly, we can derive other b_i's. From (3.1) and (3.4) we have

$$F_1 \equiv c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{17}, c_{18}$$

$$= b_1, b_3, b_5, b_7, b_9, b_{11}, b_{13}, b_{15}$$

$$F_4 \equiv c_{41}, c_{42}, c_{43}, c_{44}, c_{45}, c_{46}, c_{47}, c_{48}$$

$$= b_1+3*b_2, b_3+3*b_4, b_5+3*b_6, b_7+3*b_8, b_9+3*b_{10},$$

$$b_{11}+3*b_{12}, b_{13}+3*b_{14}, b_{15}+3*b_{16} \qquad (3.7)$$

Finally, by using the result in (3.6) we form a character string from F_1 and F_4 as follows:

$$c_{11}, (-c_{11}+c_{41})/3, c_{12}, (-c_{12}+c_{42})/3, c_{13}, (-c_{13}+c_{43})/3, c_{14}, (-c_{14}+c_{44})/3,$$

$$c_{15}, (-c_{15}+c_{45})/3, c_{16}, (-c_{16}+c_{46})/3, c_{17}, (-c_{17}+c_{47})/3, c_{18}, (-c_{18}+c_{48})/3$$

$$= b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, b_{15}, b_{16}$$

$$= F \text{ (the original file)}$$

Now, by using modulo arithmetic we will prove that b_2 can be uniquely determined by c_{11} and c_{41}. To simplify the notation (e.g., use c instead of c_{41}), we will prove the following equivalent theorem.

Theorem 1.

Given $0 <= c, b_1, b_2 <= 255$, and $c \equiv b_1 + 3 * b_2$ (modulo 256),

we want to prove that b_2 is uniquely determined by c and b_1.

Proof. By definition of modulo arithmetic, we have the following equation:

$$c = b_1 + 3 * b_2 + 256x \qquad (I.1)$$

where x is an integer. Assume that we have another integer value p

($0 <= p <= 255$) which satisfies the same modulo equation, i.e.,

$$c = b_1 + 3 * p + 256y \qquad (I.2)$$

where y is also an integer.

From (I.1) we get $3 * b_2 = c - b_1 - 256x$ (I.3)

From (I.2) we get $3 * p = c - b_1 - 256y$ (I.4)

(I.4) - (I.3): $3(p - b_2) = 256(x - y)$

Divided by 3 both sides: $p - b_2 = 256 (x - y) / 3$

Because $p - b_2$ is an integer, x - y must be a multiple of 3, hence

$$p - b_2 = 256m \qquad (I.5)$$

where m is an integer.

If m is not zero, the distance length between p and b_2 is at least 256 from eq. (I.5), this contradicts the following:

$-255 <= p - b_2 <= 255$ from the given conditions.

Therefore m must be zero, from (I.5) we then have $p = b_2$, and b_2 is uniquely determined.

This concludes theorem 1.

3.3.2.3 Similarly, we can construct the original file F if we have received any other two of the four messages F_1, F_2, F_3 and F_4, and we can have a storage more than 8 bits for each character of F_4. Because, if we use modulo 256 arithmetic and the same 8-bit storage for each character of F_4, we may not be able to uniquely determine the characters of the original file. In particular, if we have received F_3 and F_4, then

$$c_{31} \equiv b_1 + b_2 \text{ (modulo 256)}$$

$$c_{41} \equiv b_1 + 3 * b_2 \text{ (modulo 256)}$$

If $b_1 = 0$ and $b_2 = 0$ then $c_{31} = c_{41} \equiv 0$ (mod 256).

If $b_1 = 128$ and $b_2 = 128$ then $c_{31} = c_{41} \equiv 0$ (mod 256) also.

Therefore b_1 and b_2 cannot be uniquely determined by the characters of F_3 and F_4.

From now on, we reserve 9-bit storage for each character of the message F_4 in the case of m=2 and n=4, and we will use modulo 512 arithmetic for F_4; other messages will still use 8-bit storage for their characters and modulo 256 arithmetic.

3.3.3 Some Other Proofs

Now, we redesign the four row vectors for m=2 and n=4, and check if everything works well.

$$\mathbf{a}_1 = (1, 0)$$

$$\mathbf{a}_2 = (0, 1)$$

$$\mathbf{a}_3 = (1, 1)$$

$$\mathbf{a}_4 = (1, 2)$$

To prove that we can recover the characters of the original file F if we have received two messages F_1 and F_4, we need theorem 2.

Theorem 2.

Given $0 <= b_1, b_2 <= 255, 0 <= c <= 511$

and $c \equiv b_1 + 2 * b_2$ (modulo 512),

we want to prove that b_2 can be uniquely determined by c and b_1.

Proof. By definition of modulo arithmetic, we have the following equation:

$$c = b_1 + 2 * b_2 + 512x \qquad \text{(II.1)}$$

where x is an integer.

Assume that we have another integer value p ($0 <= p <= 255$) which satisfies the same modulo equation, i.e.,

$$c = b_1 + 2 * p + 512y \qquad \text{(II.2)}$$

where y is also an integer.

From (II.1) we get $2 * b_2 = c - b_1 - 512x$ (II.3)

From (II.2) we get $2 * p = c - b_1 - 512y$ (II.4)

(II.4) - (II.3): $2(p - b_2) = 512(x - y)$

Divided by 2 both sides: $p - b_2 = 256 (x - y)$

$$p - b_2 = 256m \qquad \text{(II.5)}$$

where m is an integer.

If m is not zero, the distance length between p and b_2 is at least 256 from eq. (II.5), this contradicts the following:

$0 <= p, b_2 <= 255$ from the given conditions.

Therefore m must be zero; from (II.5) we then have $p = b_2$, hence b_2 is uniquely determined.

This concludes theorem 2.

To show that we can recover the characters of the original file if we have received only two messages F_3 and F_4, we need theorem 3.

Theorem 3.

Given $0 <= b_1, b_2, c_1 <= 255$, $0 <= c_2 <= 511$ and

$b_1 + b_2 \equiv c_1 \pmod{256}$ i.e., $b_1 + b_2 + 256x = c_1$ (III.1)

$b_1 + 2b_2 \equiv c_2 \pmod{512}$ i.e., $b_1 + 2b_2 + 512y = c_2$ (III.2)

where x, y are integers,

we want to prove that b_1 and b_2 can be uniquely determined by c_1 and c_2.

Proof. Assume that we have other integers p and q ($0 <= p, q <= 255$) to satisfy the above modulo equations, i.e.,

$p + q + 256x' = c_1$ (III.3)

$p + 2q + 512y' = c_2$ (III.4)

where x' and y' are also integers.

(III.2)-(III.1): $b_2 + 256(2y - x) = c_2 - c_1$ (III.5)

(III.4)-(III.3): $q + 256(2y' - x') = c_2 - c_1$ (III.6)

(III.6)-(III.5): $q - b_2 + 256(2y' - 2y + x - x') = 0$

Hence $q - b_2 = 256m$ where $m = -(2y' - 2y + x - x')$ is an integer.

If m is not equal to zero then $|q - b_2|$ is at least 256 and this contradicts the given conditions that $0 <= q, b_2 <= 255$.

Hence m is equal to zero, and b_2 is uniquely determined.

Since b_2 can be found, b_1 can also be uniquely determined from eq. (III.1).

Note that each character of F_1, F_2 and F_3 needs 8-bit storage, but each character of F_4 needs 9-bit storage.

This concludes theorem 3.

3.3.4 Optimum Solution for the case of m = 2 and n = 4

We now try to find an optimal solution for the case of m=2 and n=4. We still want to keep the first two row vectors because they make the dispersal and reconstruction of the original file very easy. But we want to see if there exist any other two row vectors such that using only 8-bit storage for every character, i.e., using modulo 256, we can uniquely recover the original file F if we have received any two messages among these four. Let these four row vectors to be as follows:

$\mathbf{a}_1 = (1, 0)$

$\mathbf{a}_2 = (0, 1)$

$\mathbf{a}_3 = (p, q)$

$\mathbf{a}_4 = (r, s)$

where p, q, r, s are integers.

Any of the four integers p, q, r and s cannot be an even number. For example, if s is an even number, say s = 2, we assume that F_1 and F_4 are received. Then the two different characters $b_2=0$ and $b_2=128$ will give the same c_{41} modulo value of zero because $2 * 128 \equiv 0$ (modulo 256), hence we cannot uniquely determine the original file F because we don't know whether we have received the character $b_2 = 0$ or $b_2 = 128$. Therefore p, q, r and s must all be odd numbers.

Suppose we have received only F_3 and F_4, we want to prove in theorem 4 that 8-bit storage for each character of every message is not enough to uniquely reconstruct the original file.

Theorem 4.

Suppose every character is stored in an 8-bit storage, i.e.,

$0 <= b_1, b_2 <= 255, 0 <= c_1, c_2 <= 255$ and

$p*b_1 + q*b_2 \equiv c_1 \pmod{256}$ i.e., $p*b_1 + q*b_2 + 256x = c_1$ (IV.1)

$r*b_1 + s*b_2 \equiv c_2 \pmod{256}$ i.e., $r*b_1 + s*b_2 + 256y = c_2$ (IV.2)

where p, q, r and s are odd numbers; and x, y are integers,

we want to prove that b_1 and b_2 can not be uniquely determined by c_1 and c_2.

Proof. Assume that we have other integers u and v ($0 <= u, v <= 255$) to satisfy the above modulo equations, i.e.,

$$p*u + q*v + 256x' = c_1 \quad \text{(IV.3)}$$

$$r*u + s*v + 256y' = c_2 \quad \text{(IV.4)}$$

where x' and y' are also integers.

r*(IV.1)-p*(IV.2): $(q*r - p*s)*b_2 + 256(rx - py) = c_1*r - c_2*p$ (IV.5)

r*(IV.3)-p*(IV.4): $(q*r - p*s)*v + 256(rx' - py') = c_1*r - c_2*p$ (IV.6)

(IV.6)-(IV.5): $(q*r - p*s)*(v - b_2) + 256(rx' - rx + py - py') = 0$ (IV.7)

Because a product of two odd numbers is an odd number and the difference of two odd numbers is an even number, hence q*r - p*s is an even number, and we can write it as 2*t where t is an integer.

(IV.7) becomes $2*t*(v - b_2) = 256m$ (IV.8)

where m = -(rx' - rx + py - py') is also an integer.

Hence $t*(v - b_2) = 128m$ (IV.9)

We can choose $b_1 = b_2 = 0$, $u = v = 128$ and $m = t$, then these numbers will satisfy the above equation (IV.9), and we cannot get a unique b_2 if we know c_1 and c_2.

For example, $c_1 = c_2 = 0$ will satisfy both the following two cases.

$b_1 = b_2 = 0$ to satisfy eqs. (IV.1) and (IV.2) and

$b_1 = b_2 = 128$ to satisfy eqs. (IV.1) and (IV.2) because of the following reasoning:

For $b_1 = b_2 = 128$,

(IV.1) becomes $p*b_1 + q*b_2 = 128p + 128q = 128\ (p + q)$ (IV.10)

Because p and q are both odd, the sum (p + q) is even, hence (IV.10) becomes $p*b_1 + q*b_2 = 256m$ where m is an integer.

Hence $p*b_1 + q*b_2 \equiv 0$ (modulo 256)

(IV.2) becomes $r*b_1 + s*b_2 = 128r + 128s = 128\ (r + s)$ (IV.11)

By the same reason as above, we can have the following

$$r*b_1 + s*b_2 \equiv 0 \text{ (modulo 256)}$$

This concludes theorem 4.

Because of theorem 4, we have to reserve a 9-bit storage for each character of F_4 and 8-bit storage for each character of other messages. We now check whether the following four row vectors are optimum.

$\mathbf{a}_1 = (1, 0)$

$\mathbf{a}_2 = (0, 1)$

$\mathbf{a}_3 = (1, 1)$

$\mathbf{a}_4 = (1, 2)$

Because of theorems 2 and 3, by using the above four row vectors we can uniquely determine the original file if we receive any two messages. Also the example uses minimum storage for messages because of theorem 4. Hence our choice is one of the optimum solutions.

Note that in an optimum solution for the case of m=2 and n=4, each character of F_1, F_2 and F_3 needs an 8-bit storage, but each character of F_4 needs a 9-bit storage.

3.3.5 Other Examples

3.3.5.1 For m = 4 and n = 6, we can construct $\mathbf{a}_i$'s as follows:

$\mathbf{a}_1 = (1\ 0\ 0\ 0)$

$\mathbf{a}_2 = (0\ 1\ 0\ 0)$

$\mathbf{a}_3 = (0\ 0\ 1\ 0)$

$\mathbf{a}_4 = (0\ 0\ 0\ 1)$

$\mathbf{a}_5 = (1\ 1\ 1\ 1)$

$\mathbf{a}_6 = (1\ 2\ 3\ 4)$

By inspection, a matrix which is formed by any four row vectors of the above is non-singular, i.e., the determinant of the matrix is non-zero. The proof of the non-singurality of the matrix will be given later. Here, each character of F_i, $1 \leq i \leq 5$, needs 8-bit

storage; each character of F_6 needs 10-bit storage.

3.3.5.2 For m = 8 and n = 10, we can construct $\mathbf{a}_i$'s as follows:

$\mathbf{a}_1 = (1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$

$\mathbf{a}_2 = (0\ 1\ 0\ 0\ 0\ 0\ 0\ 0)$

$\mathbf{a}_3 = (0\ 0\ 1\ 0\ 0\ 0\ 0\ 0)$

$\mathbf{a}_4 = (0\ 0\ 0\ 1\ 0\ 0\ 0\ 0)$

$\mathbf{a}_5 = (0\ 0\ 0\ 0\ 1\ 0\ 0\ 0)$

$\mathbf{a}_6 = (0\ 0\ 0\ 0\ 0\ 1\ 0\ 0)$

$\mathbf{a}_7 = (0\ 0\ 0\ 0\ 0\ 0\ 1\ 0)$

$\mathbf{a}_8 = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 1)$

$\mathbf{a}_9 = (1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)$

$\mathbf{a}_{10} = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8)$

Similarly, any eight row vectors of the above form an 8 by 8 non-singular matrix.

Here, each character of F_{10} needs 11-bit storage, and each character of other subfiles needs only 8-bit storage.

3.3.6 Rank of Matrices of the Above Examples

An efficient algorithm to check the **rank of a matrix** can be found in Ref. [Grest]. We now want to prove that any m rows from n=m+2 rows of each example above are linearly independent, hence they form a non-singular matrix.

3.3.6.1 The first m rows are of course linearly independent because they form an identity matrix.

3.3.6.2 If any one row, say a_i, $1 <= i <= m$, is replaced by the next to the last row, i.e., a_{m+1}, the ith column entry of a_{m+1} is one, therefore, it increases the rank of the matrix of the other m-1 rows by one. Therefore, these m rows are all linearly independent.

3.3.6.3 If any one row, say a_i, $1 <= i <= m$, is replaced by the last row, i.e., a_{m+2}, the ith column entry of a_{m+2} is non zero, therefore, it increases the rank of the matrix of the other m-1 rows by one. Therefore, these m rows are all linearly independent.

3.3.6.4 If any one row, say a_i, is replaced by a_{m+1}; and another row, say a_j, is replaced by a_{m+2}, where j is not equal to i. Now, we consider the ith and jth rows and the ith and jth columns of the new matrix. The entries are non zero and they form a non-singular 2 x 2 matrix, therefore, these two new rows increase the rank of other m-2 rows by two. Hence, these m rows are all linearly independent.

From the above discussion, we know that any m rows from n=m+2 rows of each example are all linearly independent and therefore they form an $m \times m$ matrix of rank m. Hence the matrix is non-singular and has a unique inverse matrix.

3.3.7 Algorithm

From the above examples, we can see how we form row vectors: for any m, we form an identity matrix m × m for the first m vectors. The next row vector $\mathbf{a}_{m+1}$ is formed with every entry to be one. The last row $\mathbf{a}_{m+2}$ is formed with an arithmetic series 1, 2, 3, ..., m. Therefore, we choose n equal to m plus two. Usually, this kind of safety margin is enough. If the user needs less safety margin, he can just use the first m+1 rows. On the other hand, if the user needs more safety margin, he can choose more row vectors which must be pairwise linearly independent with the existing rows.

Each character of F_i, $1 \leq i \leq m+1$, needs 8-bit storage; each character of F_{m+2} needs $m + \lceil \log_2 m \rceil$ bits storage[††††].

Because any m row vectors of the above examples are linearly independent as proved before, if we have received m messages from n ones, then we can use $\mathbf{a}_i$'s which correspond to these m messages to form an m × m non-singular matrix, and find the inverse of this matrix. We then multiply this inverse matrix to a column vector which is formed by extracting each character from these m messages to find the first m characters of the original file. We can again multiply this inverse matrix to a second column vector which is formed by extracting each second character from these m messages to find the next m characters of the original file, and so on.

The method of determining the original file is straight forward if we receive any m messages:

[††††] $\lceil r \rceil$ is a ceiling function of a real number r, i.e., it is the smallest integer which is not smaller than r.

3.3.7.1 If we have received all first m messages, we can disregard other message(s), all we have to do is to get each character from these m messages in sequence to form the original file.

3.3.7.2 If we have received less than m messages we cannot reconstruct the original file. Otherwise, if we miss some of the first m messages, then their **a**'s are replaced by those **a**'s corresponding to the other messages we have received. By using **Gauss elimination method**, we can find the inverse of the matrix of **a**'s easily because at most two rows need to be operated. The number of row operations for forming the inverse of a matrix is at most 3*(2m-2)+3 additions, (m-2)+4 multiplications, 2 subtractions and one division and the division is the last step. From this inverse matrix, we can find the original file F from these m messages as we mentioned previously.

3.3.8 Results of IDA Design

3.3.8.1 IDA can be used to reliably transmit and store files. The small n of 4 and m of 2 of the first example can also be used for a big file equally well: the chosen matrix of the example can be used to compute the original file from any two messages of the four which have been received. As we have seen, if we have received F_1 and F_2, other F_i's can be discarded and no computation is required to restore the original file F. In this case, all we need to do is to extract a character from F_1 and F_2 successively until the end of these two messages is reached.

3.3.8.2 IDA can be used to send a file through a network by sending its subfiles, therefore the load on the network is more balanced. If the users at the source and destination nodes choose not to disclose vectors $\mathbf{a}_i$'s, they can achieve extra security besides the usual encryption security. On the other hand, the user can include m and $\mathbf{a}_i$ in message F_i (i in F_i of course), so that the sender can dynamically vary n and m, depending on the load of the network and the safety margin he desires.

3.3.8.3 IDA can be space efficient because we can choose n/m to be a little bit larger than 1, each character of the first m+1 messages occupies 8 bits, only the characters of the (m+2)th message occupy $m + \lceil \log_2 m \rceil$ bits storage each. Also, IDA can be time efficient if we choose m, n and $\mathbf{a}_i$'s intelligently.

3.4 Summary

The LZWC algorithm is a universal data compression, i.e., it compresses various kinds of data files effectively without prior knowledge of the statistics of the data files. It is also adaptive, i.e., it can gather the statistics of the data file during compression or expansion processes. The statistics file is decided before hand. The dictionaries contain most recently used entries, and they will never overflow. On the other hand, the LZW algorithm can easily flood its string table of size 4096 entries.

By organizing the dictionaries in **hierarchical** levels, LZWC uses only one byte as a pointer to a more frequently occurring string, and two bytes as a pointer to a less frequently occurring string. As long as the most significant bit of a pointer is not

corrupted, bit corruption will be contained and thus will not easily propagate. As the example of compressing the Gettysburg Address shows, even though the text file is short (1473 bytes), an impressive compression result is obtained. By comparing the popular shareware program PKZIP/PKUNZIP (an LZW implementation) with the LZWC algorithm, we know that the performance of the latter is significantly better than that of the former. Unlike adaptive Huffman and adaptive arithmetic codings, the LZWC algorithm needs to update at most one branch of the tree of each dictionary when the statistics of any entry changes substantially. Since the LZWC algorithm is universal, the precious short dictionary can contain some special symbols for other data files. Other data files, such as C programs, can also be effectively compressed.

This thesis describes 2-level hierarchical dictionaries of fixed sizes as an example. Other variations of the LZWC algorithm can also be investigated. Three of them are suggested as follows:

3.4.1 The first variation of the LZWC algorithm: The short dictionary still has 128 entries and thus needs 7 bits as its address, but the long dictionary now has 8192 entries and needs 13 bits as its address. If we try this variation of the LZWC algorithm to compress the Gettysburg Address, and we assume the words are still in the dictionaries (a reasonable assumption), then every pointer to the long dictionary has a length of 13 bits instead of 15, then we need 14 bits for each code instead of 16 for the long dictionary. This results 310 + 161 * 2 * (14/16) = 592 bytes of storage. Now, the compression ratio is 1473/592 = 2.49, and the compression percentage is 592/1473 =

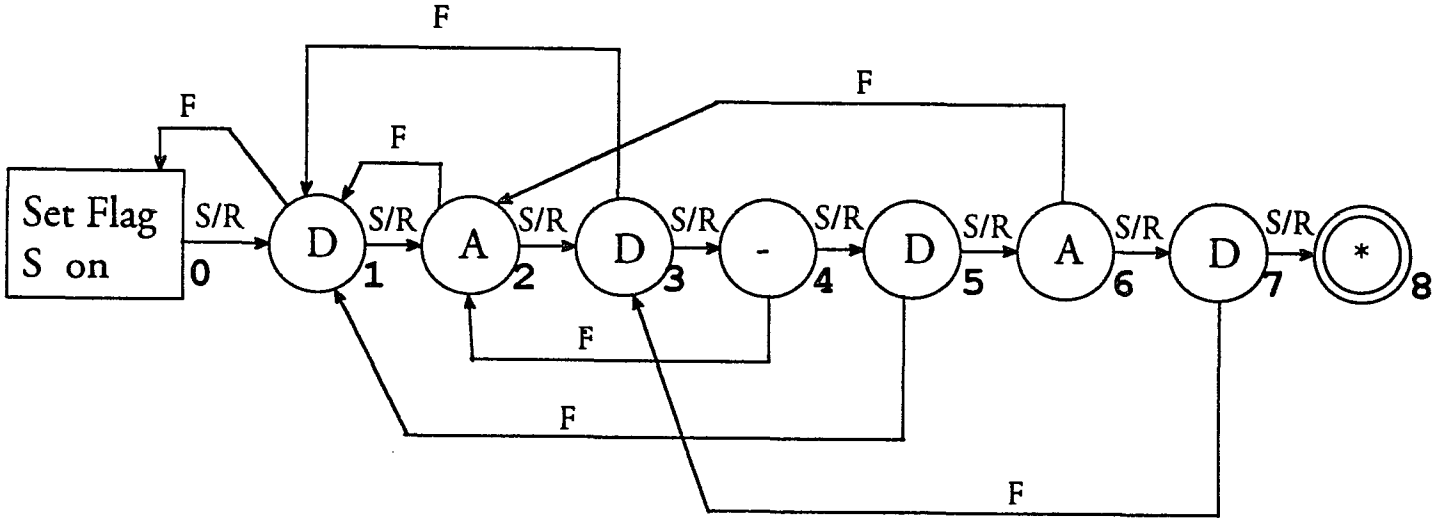
40.2%.

3.4.2 The second variation of the LZWC algorithm: The short dictionary still has 128 entries and needs 7 bits as its address, but the long dictionary initially has 8192 entries and needs 13 bits as its address. The compressor (as well as the expander) will decide whether a longer dictionary, say a dictionary of 32768 entries, will be used in future compression/expansion after the long dictionary is filled.

3.4.3 The third variation of the LZWC algorithm: There are 3-level hierarchical dictionaries. The first dictionary has 8 entries and has 3 bits as its address. The second dictionary has 64 entries and has 6 bits as its address. The third dictionary has 16384 entries and has 14 bits as its address. The code generated by the compressor determines which dictionary it points to. If the most significant bit of a code is zero, then the next 3 bits are used as a pointer to the first dictionary, else if the second most significant bit is zero, then the next 6 bits are used as a pointer to the second dictionary else the next 14 bits are used as a pointer to the third dictionary.

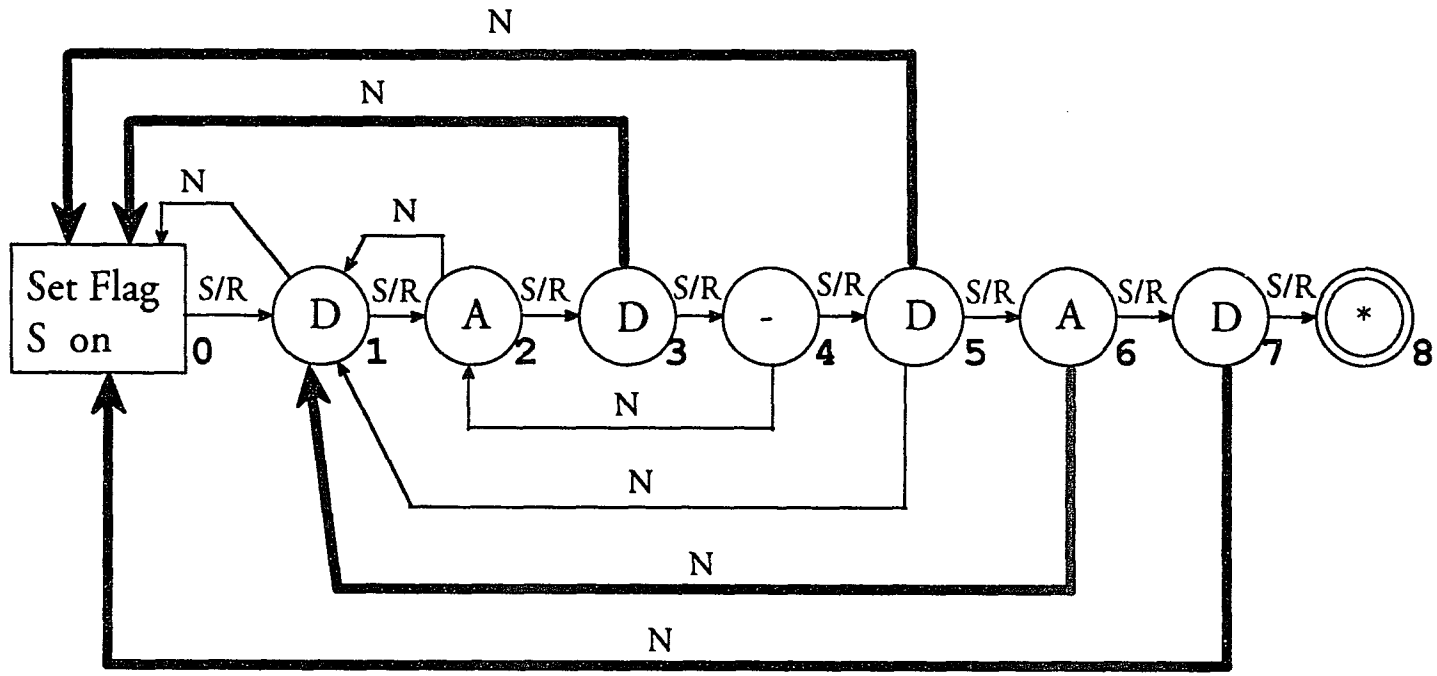
Appendix

Fig. 2 Knuth-Morris-Pratt Automaton for Pattern "DAD-DAD"



Pattern:	D	A	D	-	D	A	D
j:	1	2	3	4	5	6	7
Fail[j]:	0	1	1	2	1	2	3
Next[j]:	0	1	0	2	0	1	0

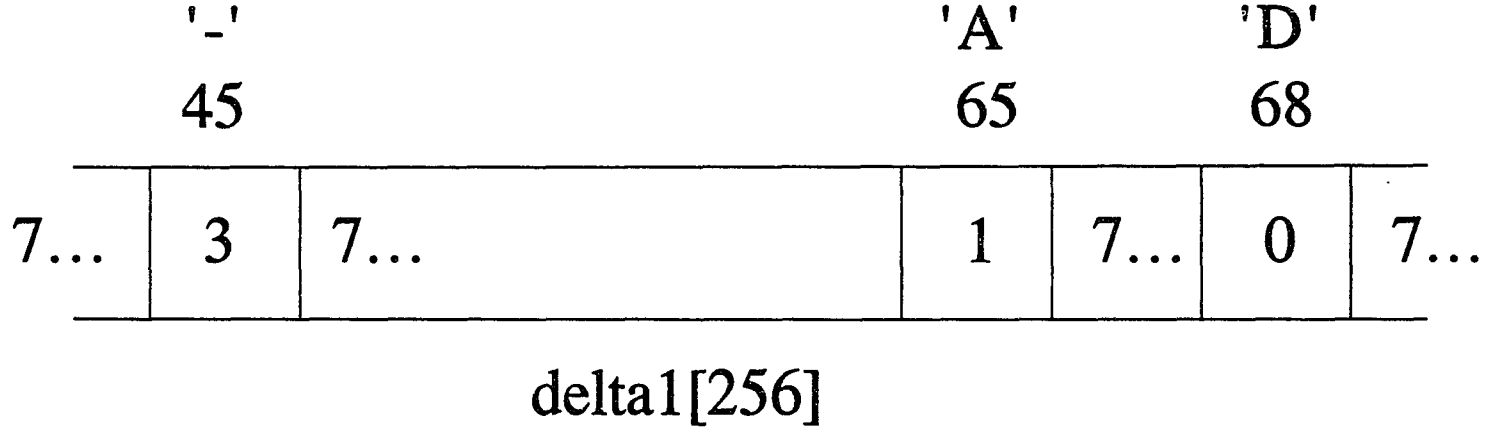
Fig. 3 Improved Knuth-Morris-Pratt Automaton for Pattern "DAD-DAD"



Pattern:	D	A	D	-	D	A	D
j:	1	2	3	4	5	6	7
Fail[j]:	0	1	1	2	1	2	3
Next[j]:	0	1	0	2	0	1	0

FIG. 4 BOYER-MOORE ALGORITHM:

USES MAX(DELTA1,DELTA2) FOR JUMPS



	7	6	5	4	3	2	1
J:	1	2	3	4	5	6	7
PATTERN:	D	A	D	-	A	A	D
DELTA2[J]:	12	11	10	9	6	7	1
PATTERN:	D	A	D	-	D	A	D
DELTA2[J]:	10	9	8	7	8	7	1

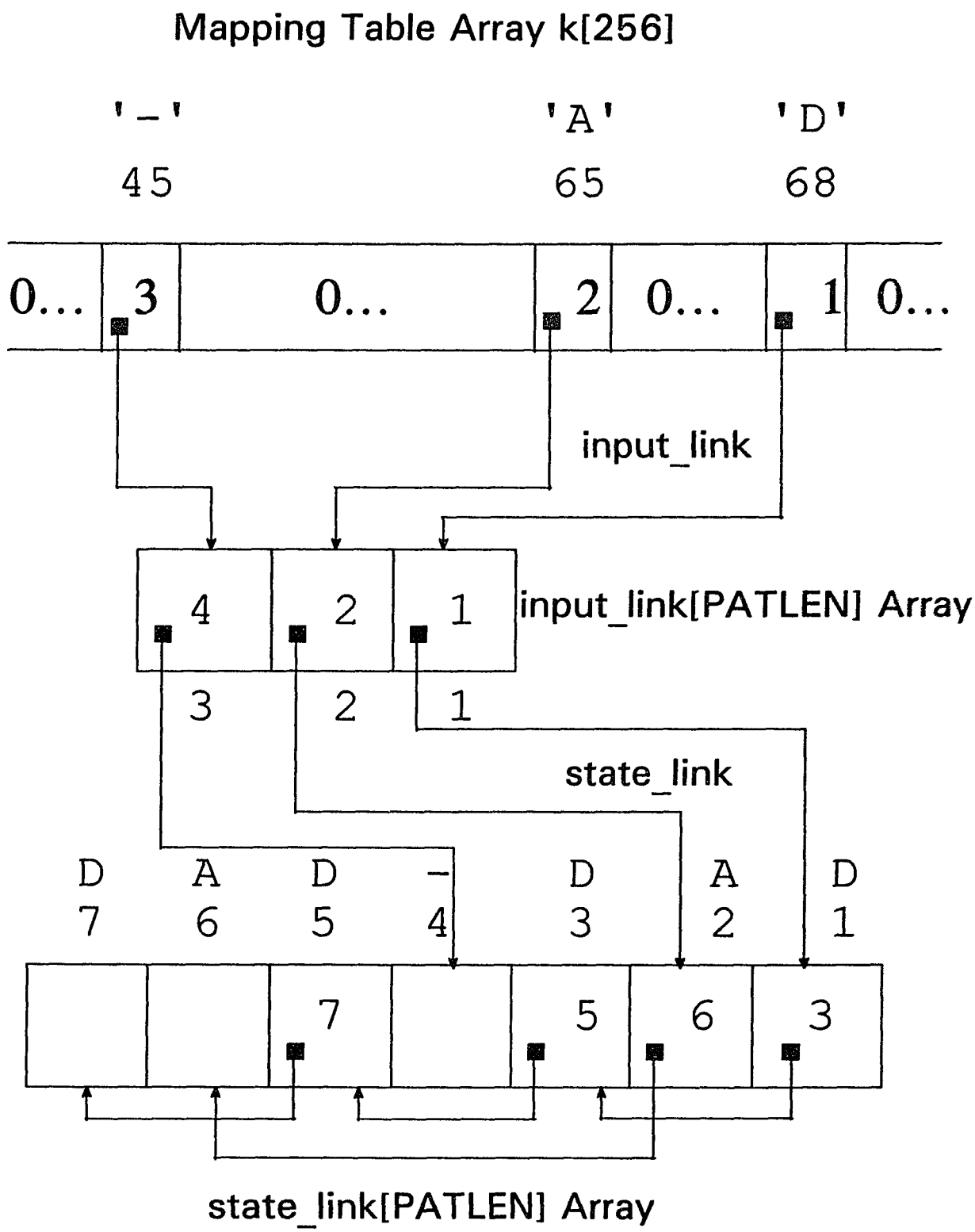


Fig. 5 Data Structure of the Three Arrays

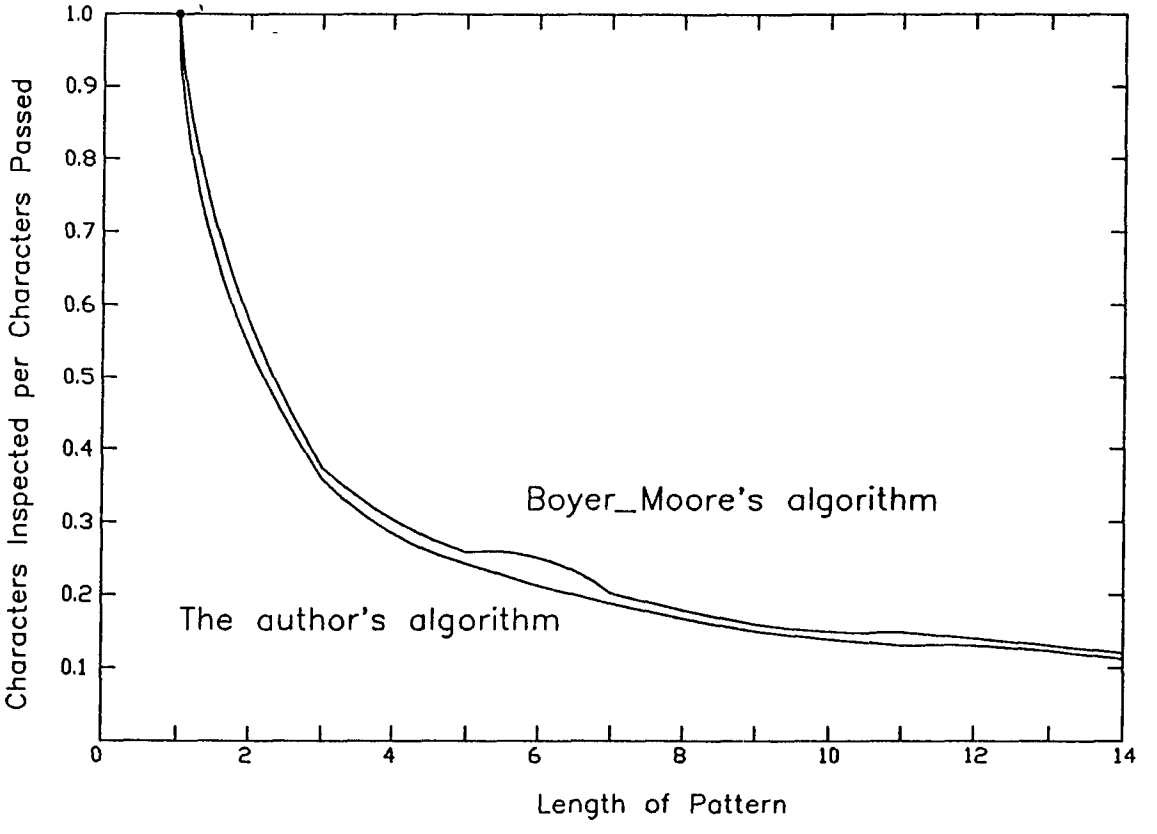


Fig. 6 No. of Characters Inspected per No. of Characters Passed

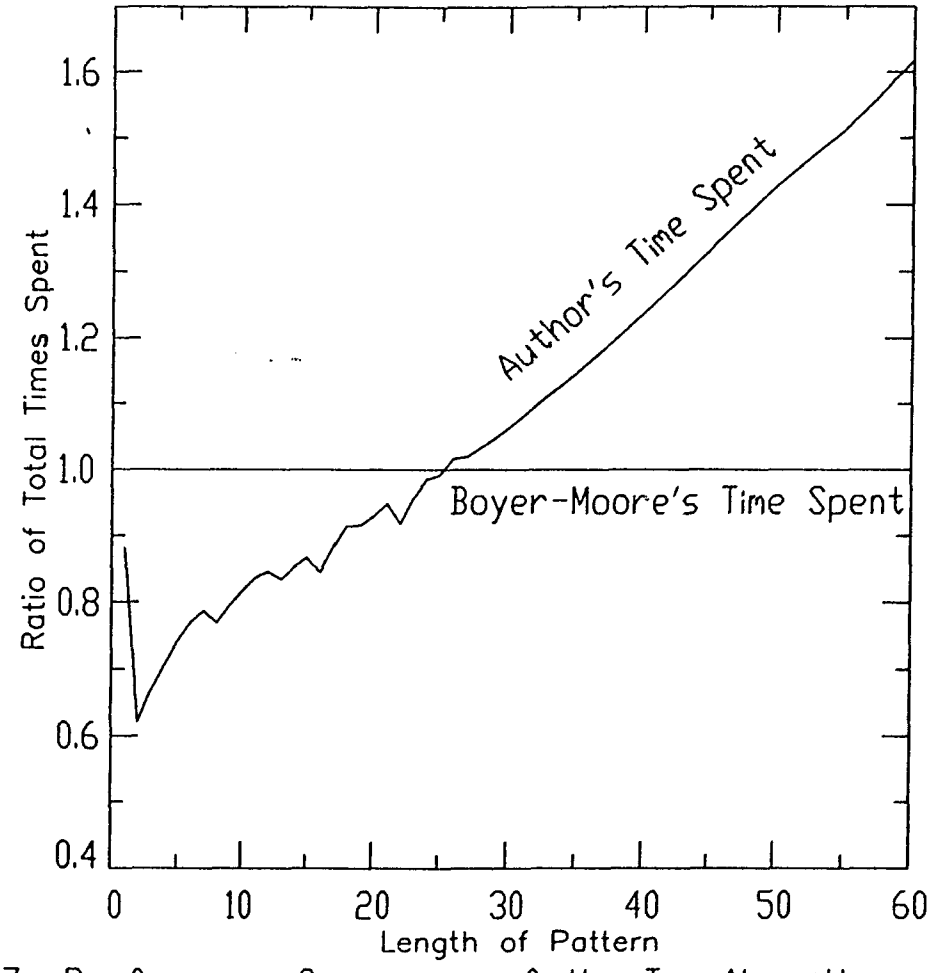


Fig. 7 Performance Comparison of the Two Algorithms

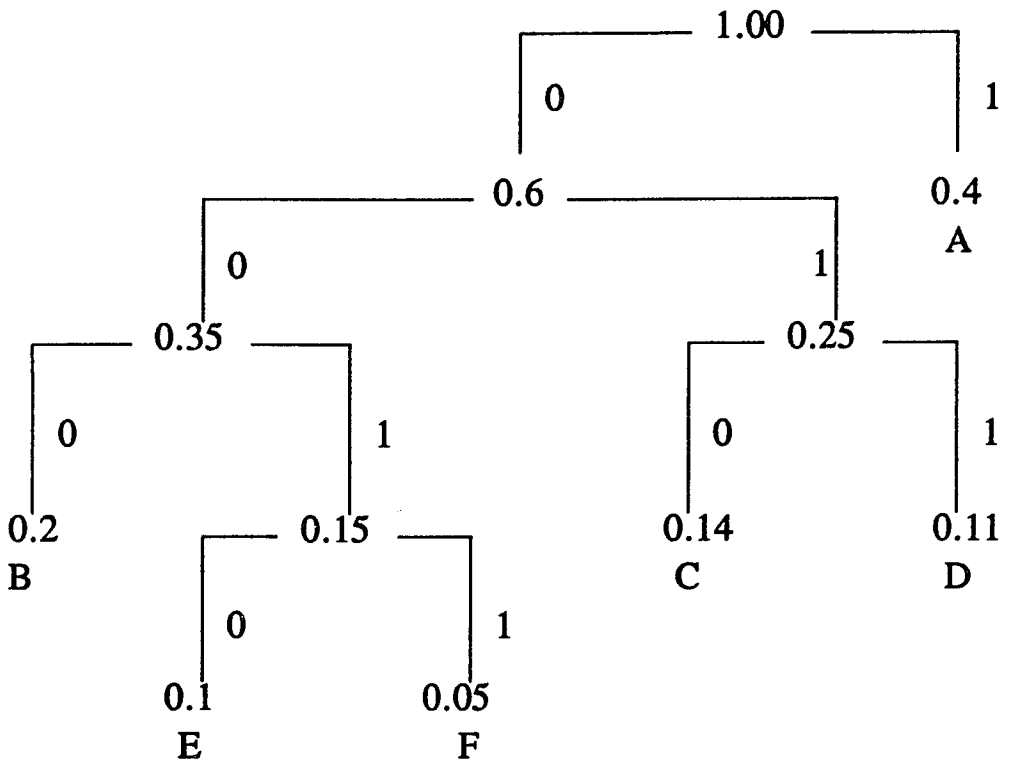


Fig. 8 Huffman Code Tree

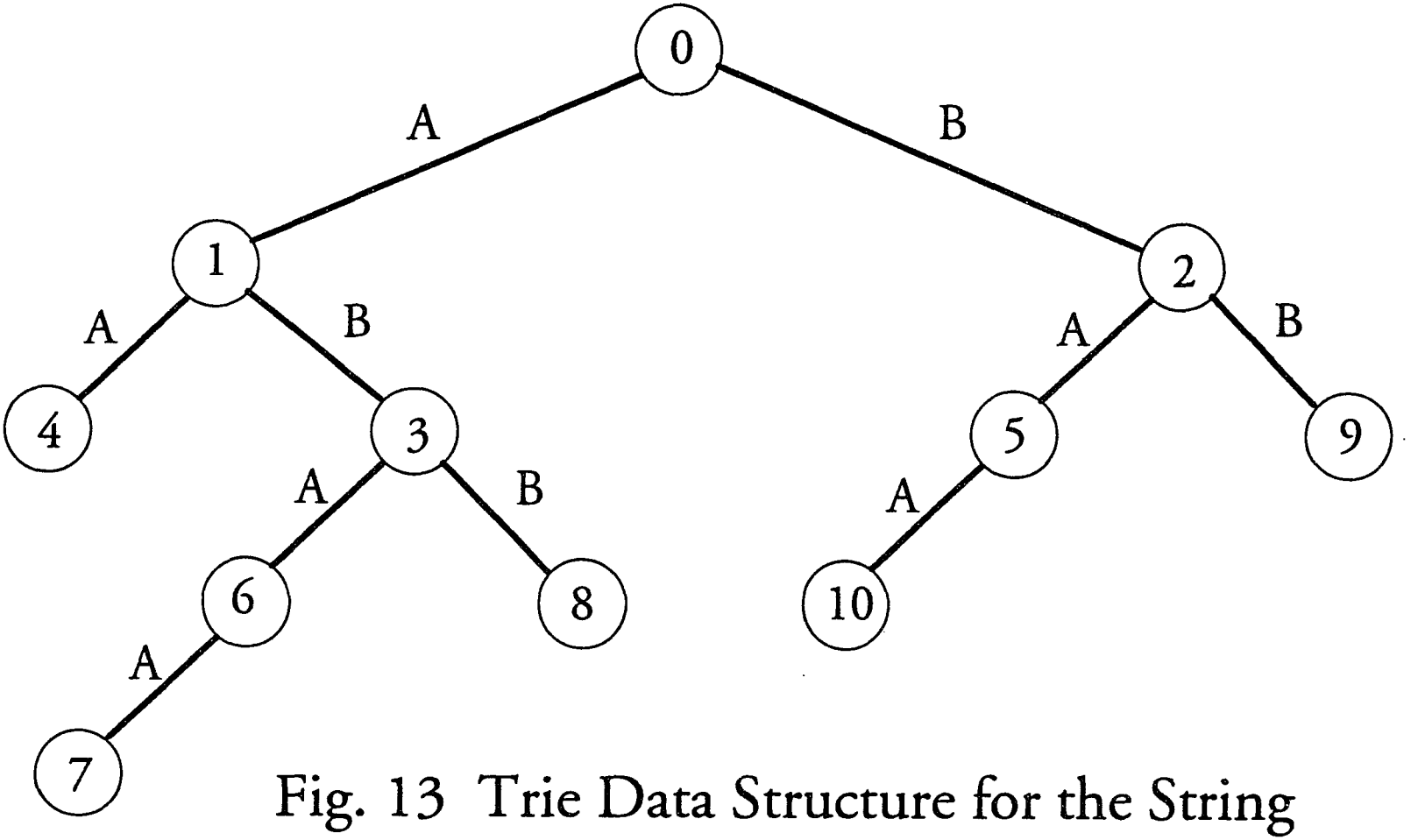


Fig. 13 Trie Data Structure for the String "A B AB AA BA ABA ABAA ABB BB BAA"

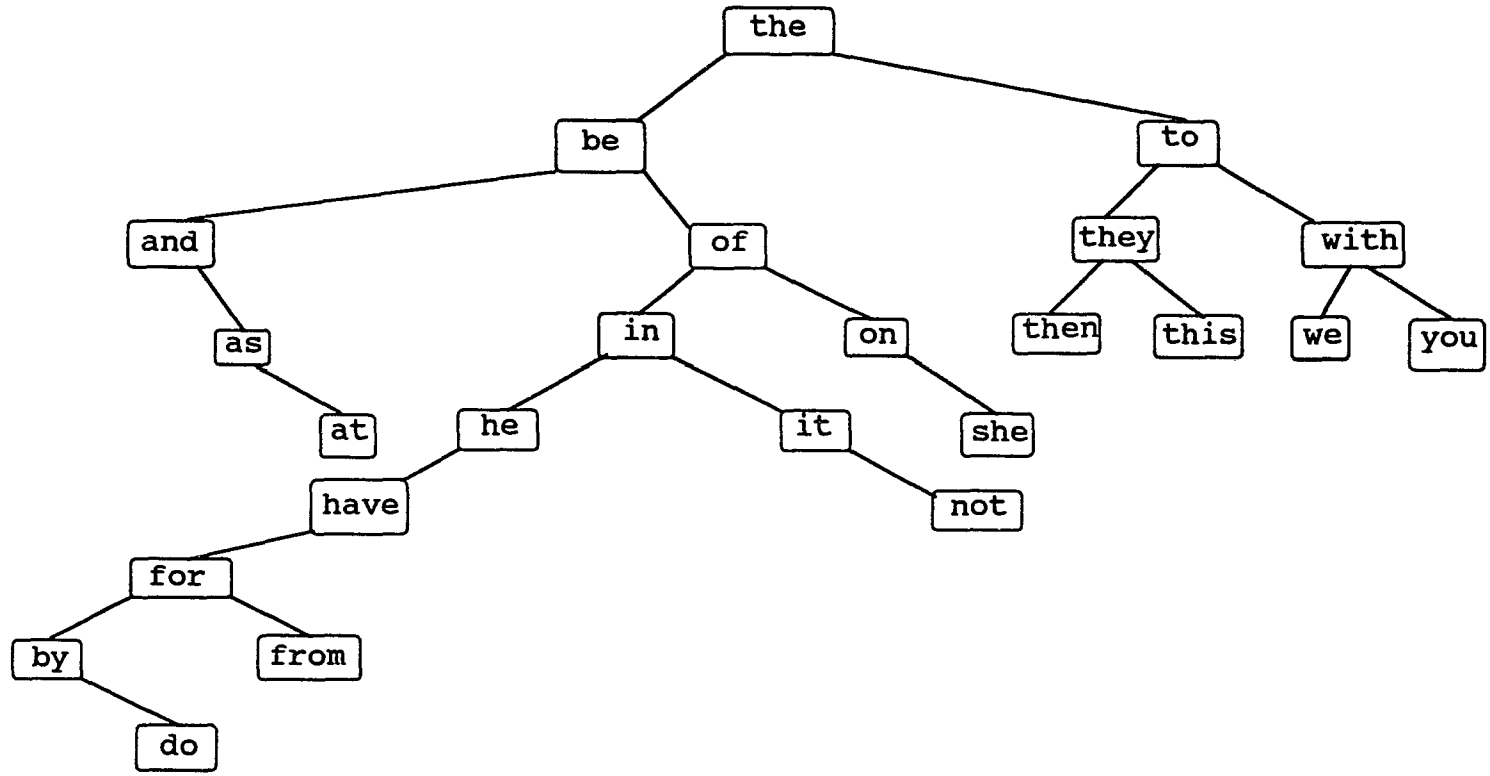
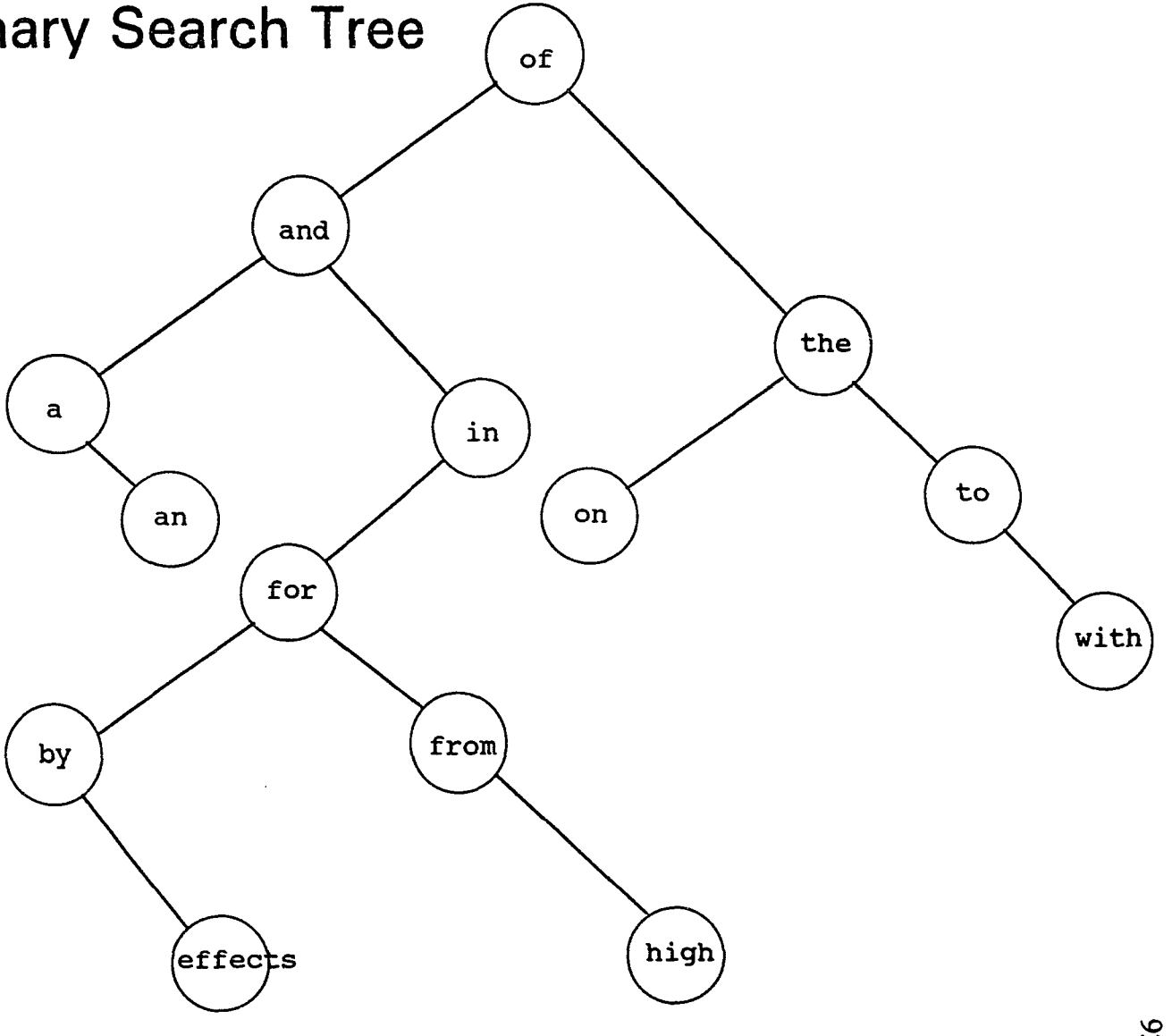


Fig. 14 Tree Structure of a Part of the Long Dictionary

Fig. 15 Binary Search Tree

word	frequency
of	142
the	79
and	69
in	64
a	32
on	22
to	18
for	15
by	13
from	10
with	9
high	8
an	7
effects	6

Rank List



References

[Aho] Aho, A.V., Hopcroft J.E., and Ullman J.D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, pp. 318-361, 1974.

[Aho75] Aho, A.V. and Corasick, M.J. *Fast pattern matching: An aid to bibliographic search.* Comm. ACM 18,6, pp. 333-340, June 1975.

[Ang] Ang P.H., Ruetz P.A. and Auld D. *Video compression makes big gains.* IEEE Spectrum Oct. 1991.

[Bell] Bell T.C., Cleary J. G. and Witten I.H., *Text Compression*, Prentice-Hall, Inc. 1990.

[CCITT] CCITT, Red Book, Volume VII - *Fascicle* VII.3, Geneva 1985.

[Boyer] Boyer R.S. and Moore J.S., *A fast string searching algorithm.* Commun. ACM 20,10, pp. 762-772, OCT. 1977.

[Chang] Chang D.K. and Rootenberg J., *A Simple Description of Efficient Dispersal of Information for Fault Tolerance*, ISMM International Conference on Parallel and Distributed Computing, and System, New York, U.S.A, pp. 203-206, October 10 -12, 1990.

[Chang91A] Chang D.K., *Exact Data Compression Using Hierarchical Dictionaries*, Proceedings of Data Compression Conference April 8-11, 1991, Snowbird, Utah, U.S.A., IEEE Computer Society Press, pp. 431, 1991.

[Chang91B] Chang D.K., *Data Compression Using Hierarchical Dictionaries.* The Journal of Systems and Software. 15, 3, pp. 223-238, July 1991.

[Chang93] Chang D.K., *A String Pattern Matching Algorithm.* The Journal of Systems and Software. (to be published in 1993)

[Clear] Cleary J. G. and Witten I. H., *Data Compression Using Adaptive Coding and Partial String Matching*, IEEE Trans. Commun. vol. COM-32, no. 4, pp. 396-402, April 1984.

[Copel] Copeland J. A., *Data Compression Technique for PC Communications*, IEEE Journal on Selected Areas in Communications, vol. 7, No. 2, pp. 246-248, Feb. 1989.

[Corma] Cormack G. V., *Data Compression on a Database System*, Communications of ACM, vol. 28, No. 12, pp. 1336-1342, Dec. 1985.

[Corte] Cortesi D., *An Effective Text-Compression Algorithm*, BYTE 7,1, pp.397-403, Jan. 1982.

[LeGal] LeGall D.J., *MPEG: A video compression standard for multimedia applications.* Commun. ACM 34, 4, pp. 46-58, April 1991.

[Fiala] Fiala E. R. and Greene D. H., *Data Compression with Finite Windows*, Communications of the ACM, vol 32, no. 4, pp. 490-505, April 1989.

[Franc] Francis W. N., Kucera H. and W. Mackie A. W., *Frequency Analysis of English Usage: Lexicon and Grammar*, Houghton Mifflin Company, Boston, 1982.

[Gonza] Gonzalez R.C. and Wintz P. [1987]. *Digital Image Processing*, 2nd ed., Addison-Wesley Publishing Company.

[Grest] Grestein L. J., *A New Algorithm for Computing the Rank of a Matrix*, The American Mathematical Monthly, Vol, 95, No. 10, pp. 950-952, December 1988.

[Guiba] Guibas, L.J. and Odlyzko A.M., *A new proof of the linearity of the Boyer-Moore string searching algorithm.* SIAM J. Comput. 9,4, pp. 672-682, Nov. 1980.

[Huffm] Huffman D. A., *A Method for the Construction of Minimum-redundancy Codes*, Proc. Inst. Electr. Radio Eng. 40,9, pp. 1098-1101, Sep., 1952.

[Hunte] Hunter R. and Robinson A. H., *International Digital Facsimile Coding Standards*, Proceedings of the IEEE, vol. 68, no. 7, pp. 854-867, 1980.

[Knuth] Knuth D.E, *Optimum Binary Search Trees*, Acta Inf. 1,1, pp. 14-25, Jan. 1971.

[Knuth77] Knuth D.E., Morris J.H. and Pratt V.R., *Fast pattern matching in string.* Siam J. Comput. 6,2, pp. 323-350, June 1977.

[Kucer] Kucera H. and Francis W.N., *Computational Analysis of Present-Day American English*, Brown University Press, Providence, R.I., 1967.

[Lelew] Lelewer D.A. and Hirschberg D. S., *Data Compression*, ACM Computing Surveys, Vol. 19, No. 3, pp. 261-296, September 1987.

[Liou] Liou M.L., *Overview of the px64 kbps video coding standard.* Commun. ACM 34,4, pp. 59-63, April 1991.

[McCre] McCreight E.M., *A Space-Economical Suffix Tree Construction Algorithm*, J. ACM 23,2, pp. 262-272, April 1976.

[Mille] Miller V.S. and Wegman M.N., *Variations on a theme by Ziv and Lempel*, IBM Res. Rep. RC 10630 (#47798), 7/31/84.

[Morri] Morrison D.R., *PATRACIA-Practical Algorithm To Retrieve Information Coded In Alphanumeric*, J. ACM 15, 4, pp. 514-534, Oct. 1968.

[Pechu] Pechura M., *File Archival Techniques Using Data Compression*, Communications of the ACM, vol. 25, no. 9, pp. 605-609, Sept. 1982.

[Rabin] Rabin M.O., *Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance*, Journal of ACM, Vol. 36, No. 2, pp. 335-348, April 1989.

[Rao] Rao K.R. and Yip P., *Discrete Cosine Transform - Algorithms, Advantages, Applications*. Academic Press, Inc, London, 1990.

[Smit] Smit G.V., *A comparison of three string matching algorithms*. Softw. - Prac. and Exp. 12, 1, pp. 57-66, Jan. 1982.

[Tanen] Tanenbaum A.S., *Computer Networks*, 2nd edition. Prentice Hall, Englewood Cliffs, N.J., 1988.

[Tropp] Tropper R., *Binary-Coded Text, A Text-Compression Method*, BYTE 7,4, pp. 98-413, April 1982.

[Vaugh] Vaughan-Nichols S.J., *Saving Space*, BYTE, pp. 237-243, March 1990.

[Walla] Wallace G.K., *The JPEG Still Picture Compression Standard*. Commun. ACM 34, 4, pp. 30-44, Apr. 1991.

[Welch] Welch T.A., *A technique for high performance data compression* Computer, vol. 17, no. 6, pp. 8-19, 1984.

[Witte] Witten I.H., Neal R. M. and Cleary J. G., *Arithmetic Coding for Data Compression*, Communication of the ACM, vol. 30, no. 6, pp. 520-540, June 1987.

[Ziv] Ziv J., "Coding Theorems for Individual Algorithm for Sequences", IEEE Trans. Info. Theory IT-24, 4, pp. 405-412, 1978.

[Ziv77] Ziv J. and Lempel A., *A Universal Algorithm for Sequential Data Compression*, IEEE Trans. Info. Theory IT-23,3, pp. 337-343, May 1977.

[Ziv78] Ziv J. and Lempel A., *Compression of Individual Sequences Via Variable-Rate Coding*, IEEE Trans. Info. Theory IT-24,5, pp. 530-536, Sept. 1978.

Autobiography

Prior to studying for his Ph.D. at the City University of New York, Daniel Kuo-Yee Chang had worked as a member of technical staff at AT&T Bell Laboratories for several years. He received a M.S.(Computer Science) from Columbia University of New York City, a Postgraduate Diploma in Systems Analysis from the Chinese University of Hong Kong, and a B.Sc.(Eng.) from University of Hong Kong. His correspondence address is 227-12 53rd Avenue, Bayside, NY 11364, U.S.A.