The IJICS (International Journal of Informatics and Computer Science)
Vol 3 No 1, March 2019
ISSN 2548-8384 (online)
ISSN 2548-8449 (print)
Page: 1 - 4

Analysis Of Database Record Compression Using The Levenstein Algorithm

Rian Syahputra

Department of Computer Science STMIK Budi Darma, North Sumatra, Indonesia

Abstract – Data stored in the database affects how much storage memory is used. The more data stored, the more memory will be used. To reduce memory usage it is necessary to reduce the data to be stored. The technique that can be used is data compression. Levenstein algorithm is a data compression technique that replaces the initial bits of the string, resulting in smaller compression. Levenstein algorithm is suitable for compressing data that has many repetitions.

Keywords– Levenstein, Compression, Record Database

1. INTRODUCTION

Databases are commonly used to store information from an application system whether it is website based or desktop based. Applications that use databases are of course aimed at utilizing existing storage capacity so that it can be accessed anytime and anywhere. But the larger the size of the data stored it requires large storage media [1], [2]. The size of the data also affects the speed of data transmission. Large data sizes are also influenced by the number of repetitions of letters or words [3]. Techniques that can be used are data compression, which allows reducing repetition of letters or words. One of the data compression algorithms is Levenstein.

2. THEORY

2.1 Compression

Compression is the process of converting data consisting of a collection of characters into a coded form that aims to save storage and time of data transmission [1]. Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, the bit stream, or the compressed stream) that has a smaller size [4]–[6].

2.2 Levenstein Code

Levenstein algorithm is a universal coding for non-negative integers developed by Vladimir Levenstein in 1968. Levenstein algorithm has multistep processes to get the Levenstein code. Levenstein code of zero is a single 0, and to code a positive number *n*, here are the steps [4]:

1. Set the count variable *C* to 1. Initialize the code-so-far to the empty string.
2. Take the binary value of *n* without its leading 1 and prepend it to the code-so-far.
3. Let *M* be the number of bits prepended in step 2.
4. If $M \neq 0$, increment *C* by 1, go to and execute step 2 with *M* instead of *n*.
5. If $M = 0$, prepend *C* 1's followed by a 0 to the code-so-far and stop.

Table 1. Levenstein Code

n	Levenstein Code
0	0
1	10
2	110 0
3	110 1
4	1110 0 00
5	1110 0 01
6	1110 0 10

The steps for compression using Levenstein are as follows [7]:

1. Sort of characters that have the largest frequency (which has the same value) to the smallest. If it has same frequency sort it by alphabetical.

ISSN 2548-8384 (online)
ISSN 2548-8449 (print)
Page: 1 - 4

2. Form a Levenstein code table. The data bits to be compressed are replaced with bits in the Levenstein code table. After replacing count the number of bits in each value.
3. Doing the Levenstein code bit string results into new data values. But before making the Levenstein code bit string results into a new data value, check the bit string length first. Following are the steps in checking the length of the bit string.
    a. If the remainder for the length of the string bit to 8 is 0 then add 00000001. State the final bit.
    b. If the remainder for the length of the bit string to 8 is n (1, 2, 3, 4, 5, 6, 7) then add 0 as much as 7 - n + "1" at the end of the bit string. Express it with L. Then add the binary number from 9 - n. State with the final bit.

The steps of decompression using the Levenstein Code algorithm are as follows:

1. Read data that has been compressed.
2. The result of the Levenstein code bit string that becomes the new data value is changed back to binary form.

Returns the binary to the original bit string by reading the last 8 bits, the reading is a decimal number. State the reading result with n then remove the bit at the end as much as 7 + n.

3. RESULT AND DISCUSSION

For example, we have input string “characters”. The first step is to sort the characters frequency by the largest to the smallest, see table 2.

Table 2. Sorting the input string

No.	Character	Frequency	Binary	Bit	Frequency x Bit
1	a	2	0110 0001	8	16
2	c	2	0110 0011	8	16
3	r	2	0111 0010	8	16
4	e	1	0110 0101	8	8
5	h	1	0110 1000	8	8
6	s	1	0111 0011	8	8
7	t	1	0111 0100	8	8
				Total	80

From the table 2 we know that the string “characters” has 80 bit size. Now, we change the binary with the Levenstein code from table 1:

Table 3. Compression with Levenstein Code

n	Character	Frequency	Levenstein Code	Bit	Frequency x Bit
0	a	2	0	1	2
1	c	2	10	2	4
2	r	2	110 0	4	8
3	e	1	110 1	4	7
4	h	1	1110 0 00	7	7
5	s	1	1110 0 01	7	7
6	t	1	1110 0 10	7	7
				Total	42

As we can see from table 3 that after we change the input binary code with Levenstein code it can compressed the input string to 42 with compression ratio is 52,50% (size before compression divide by size after compression), this means we have 52,5% data left after the compression using Levenstein. From table above we can make the new table data in the database that will be used to compare code during the decompression process.

Now we arrange the new codes as string bit for “characters”

c	h	a	r	a	c	t	e	r	s
10	1110000	0	1100	0	10	1110010	1101	1100	1110001

From the string bit above we get 39 bit, then we will look for the remainder of 39 for 8 and we get 7 as the remainder for and state as *n*. Because the remainder for the length of bit string is 7, then add 0 as much as 7 - *n* + "1" = 7 – 7 + "1" = "1" and state it with L, from that we add padding as much as "1" bit at the end of the bit string as we can see below with red colored:

10111000 00110001 01110010 11011100 11100011

Then add the binary number from 9 - *n* = 9 – 7 = 2 = 00000010 state it with final bit and add to the end of the string bit after padding. And we get the final string bit as we can see below with red colored:

10111000 00110001 01110010 11011100 11100011 00000010

Now we finish compression data with Levenstein. Then we generate per bit to look for new character we get after compression. See table 4.

Table 4. Generating new code

String Bit	New Character
10111000	¸
00110001	1
01110010	r
11011100	Ü
11100011	ã
00000010	STX

Then the new character after the compression process will be stored in the database.

Decompression of Levenstein Code are as following:

1. First, read the record from database.
2. Then changed back to binary form.

   10111000 00110001 01110010 11011100 11100011 00000010

3. Returns the binary to the original bit string by reading the last 8 bits and convert it to decimal number.

   00000010 = 2

   State the reading result with *n* then remove the bit at the end as much as 7 + *n* = 7 + 2 = 9, we remove 9 bit from the last bit and we get the string bit as following:

   10111000 00110001 01110010 11011100 1110001

   From the string bit above we read the bit from left to right one by one and compare it with table 3. Read first index-0 from the string bit is 1 then compare it with table 3. If not available, then join with the next index, we get index-0 + index-1 are = 10 then compare it with table 3. If available, change the bit with the found "c". And then we read the next index is index-2 is 1 then compare it with table 3, and not available, so we continue the read index-2 + index-3 are 11 then compare it with table 3, and not available, continue read index-2 + index-3 + index-4 are 111 and so on … we get in index-2 + index-3 + index-4 + … + index-7 are 111000 and compare it with table 3 we found "h". And so on until all bit strings are replaced with the initial character.

10	1110000	0	1100	0	10	1110010	1101	1100	1110001
c	h	a	r	a	c	t	e	r	s

4. CONCLUSION

The conclusions we can summarized are as follows:

ISSN 2548-8384 (online)
ISSN 2548-8449 (print)
Page: 1 - 4

1. The greater the frequency of appearance of the character the smaller the compression results.
2. The disadvantage of this research is having to create a new table in the database to store the Levenstein code that is used to change the compression character to the initial character. Making this new table will increase memory capacity.
3. Levenstein algorithm is suitable for compressing data that has many repetitions.

REFERENCES

[1] S. D. Nasution, "PERANCANGAN APLIKASI KOMPRESI FILE TEKS DENGAN MENERAPKAN ALGORITMA GOLDBACH CODES," *J. Ilm. INFOTEK*, vol. 1, no. 1, pp. 104–109, 2013.
[2] S. D. Nasution and Mesran, "Goldbach Codes Algorithm for Text Compression," *IJournals Int. J. Softw. Hardw. Res. Eng.*, vol. 4, no. December, pp. 43–46, 2016.
[3] E. Buulolo, "Sequitur Algorithm For Particular Character Compression Or Words Always Returned," *Int. J. Informatics Comput. Sci. (The IJICS)*, vol. 1, no. 1, pp. 15–17, 2017.
[4] D. Salomon, D. Bryant, and G. Motta, *Handbook of Data Compression (Google eBook)*, Fifth. Springer, 2010.
[5] M. R. Irliansyah, S. D. Nasution, and K. Ulfa, "Penerapan Metode Deflate Dan Algoritma Goldbach Codes Dalam Kompresi File Teks," *KOMIK (Konferensi Nas. Teknol. Inf. dan Komputer)*, vol. 1, no. 1, pp. 186–189, 2017.
[6] S. D. Nasution, G. L. Ginting, M. Syahrizal, and R. Rahim, "Data Security Using Vigenere Cipher and Goldbach Codes Algorithm," *Int. J. Eng. Res. Technol.*, vol. 6, no. 01, pp. 360–363, 2017.
[7] T. P. Sari, S. D. Nasution, and R. K. Hondro, "Penerapan Algoritma Levenstein Pada Aplikasi Kompresi File Mp3," *KOMIK (Konferensi Nas. Teknol. Inf. dan Komputer)*, vol. 2, no. 1, 2018.