A NEW APPROACH TO DICTIONARY-BASED LOSSLESS COMPRESSION

Altan MESUT
altanmesut@trakya.edu.tr
Computer Engineering Department
Trakya University, TURKEY

Aydın CARUS
aydinc@trakya.edu.tr
Computer Engineering Department
Trakya University, TURKEY

Abstract

In this paper, a new approach on dictionary-based lossless compression method is introduced. In our two-pass compression algorithm (SSDC), most frequently used two character blocks (digrams) are found in source file in the first-pass, and they are inserted into free spaces in ASCII table which are unused by the document in the second-pass. In our multi-pass algorithm (RSSDC), the two-pass algorithm is called particular number of times recursively. In each iteration, "total free space / total number of iteration" of the free spaces in the table is filled. In order to increase compression ratio, we also extend the ASCII table to 512 characters, by increasing bits per character from 8 to 9.

Keywords: Digram Coding, Huffman Coding, LZ77, LZW, lossless compression.

INTRODUCTION

Data compression techniques are widely used to transfer data faster on a network and store data in less capacity on a hard drive.

Lossy data compression reduces the size of the source data by permanently eliminating certain information, especially redundant information. When the file is uncompressed, only a part of the original information is retrieved. Lossy data compression is generally used for image, video and sound, where a certain amount of information loss will not be detected by most users.

Lossless data compression is used when it is important that the original and the decompressed data should be exactly identical, or when no assumption can be made on whether certain deviation is uncritical. Typical examples are text documents, executable programs and source code.

There are two types of lossless compression techniques; statistical-based and dictionary-based. In statistical-based techniques, compression takes place based on the frequency of input characters. The most well known statistical-based techniques are; Huffman Coding [1, 2], and Arithmetic Coding [3, 6]. Dictionary-based techniques replace input strings with earlier identical input. We can divide dictionary-based techniques into three categories. In static dictionary scheme, the dictionary is the same for all inputs. In semi-static dictionary scheme, distribution of the symbols in the input sequence learned in the first-pass, compression of the data made in the second-pass by using a dictionary derived from the distribution learned. In adaptive (dynamic) dictionary scheme, the dictionary is a portion of the previously encoded sequence. Static dictionary is most appropriate when considerable prior knowledge about the source is available. If there is not sufficient prior knowledge about the source, using adaptive or semi-static schemes is more effective. Most adaptive dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel in 1977 [8] and 1978 [9]. The approaches based on the 1977 paper are said to belong to the LZ77 family, while the approaches based on the 1978 paper are said to belong to the LZ78 family. The most well known modification of LZ78 Algorithm is Terry Welch's LZW Algorithm [5].

DIGRAM CODING

Digram coding is a static dictionary technique that is less specific to a single application. In digram coding, the dictionary consists of all letters of the source alphabet followed by as many pairs of letters, called *digrams*, as can be accommodated by the dictionary [4].

The Digram Encoder: Reads a two-character input and searches the dictionary to see if this input exists in the dictionary. If it does, the corresponding index is encoded and transmitted. If it does not, the first character of the pair is encoded. The second character of the pair then

becomes the first character of the next digram the encoder reads another character to complete the digram, and the search procedure is repeated.

SEMI-STATIC DIGRAM CODING (SSDC)

We have developed a semi-static compression algorithm based on digram coding. It runs in two-pass by the nature of semi-static dictionary scheme.

In the first-pass of our two-pass digram coding algorithm, all of the individual characters that are used in the source file are found and they are added to the dictionary. In addition, all of the pairs of characters and the number of their occurrence in the file are found and recorded. Then, the pairs are sorted according to number of their occurrence. If the source file contains *n* individual characters, and the dictionary size is *d*, then the number of digrams that can be added to the dictionary is *d-n*. Thus, the first *d-n* pairs that are the most frequently occurred in the file are chosen, and the rest of the dictionary is filled with them. Before the second-pass, the *n* value is written to the beginning of the destination file. After the *n* value, the dictionary that contains n individual characters and d-n digrams is written.

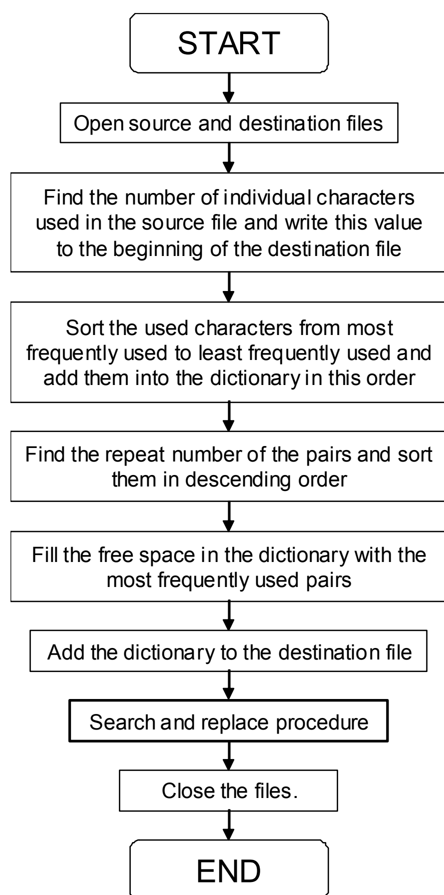

Figure 1. Flowchart of SSDC

In the second-pass, our search and replace procedure do what the digram encoder do. The procedure starts from beginning of the source file and reads two-character to form the digram. It searches the digram in the dictionary. If the digram exists in the dictionary, the corresponding index is written to the destination file. If it does not, the first character of the pair is written. The flowchart of this algorithm is shown in Figure 1.

The one-pass decompression algorithm is very simple and it runs much faster then the compression. Firstly, it reads a character from the beginning of the compressed file. This character represents the individual characters in the uncompressed file (the *n* value). Then it reads *n* individual characters and places them in the beginning of the dictionary. Later, it reads two characters *d-n* times for getting digrams, and places them into the dictionary. After the dictionary is regained, the reverse of the search and replace procedure is done for decompression.

RECURSIVE SEMI-STATIC DIGRAM CODING (RSSDC)

We have developed another algorithm based on our semi-static digram coding algorithm. In this second algorithm, we used a recursive approach to increase compression ratio. This multi-pass algorithm is not filled all of the free space in the dictionary in one pass. The free space is divided into the number of iterations, and each iteration fills its own free space. A digram, which is added in the nth iteration, will become a character in $(n+1)^{th}$ iteration.

For example, suppose the source file contains 86 individual characters and the dictionary size is 256. If the number of iterations is 10, each iteration adds (256 − 86) / 10 = 17 digrams in the dictionary. After the first iteration, the destination file contains 86 + 17 = 103 individual characters. For example, if "_the" is one of the most repeated character groups in the source file, "_t" and "he" pairs might be inserted in 87-103 interval of the dictionary. Suppose "_t" placed in 90, and "he" placed in 88. In the second iteration, the "90+88" pair might be one of the most frequently used digram and can be placed between 104 and 120. The flowchart of this algorithm is shown in Figure 2.

In addition to the decompression algorithm described in semi-static digram coding, this time, the reverse of the search and replace procedure is done recursively, as shown below:

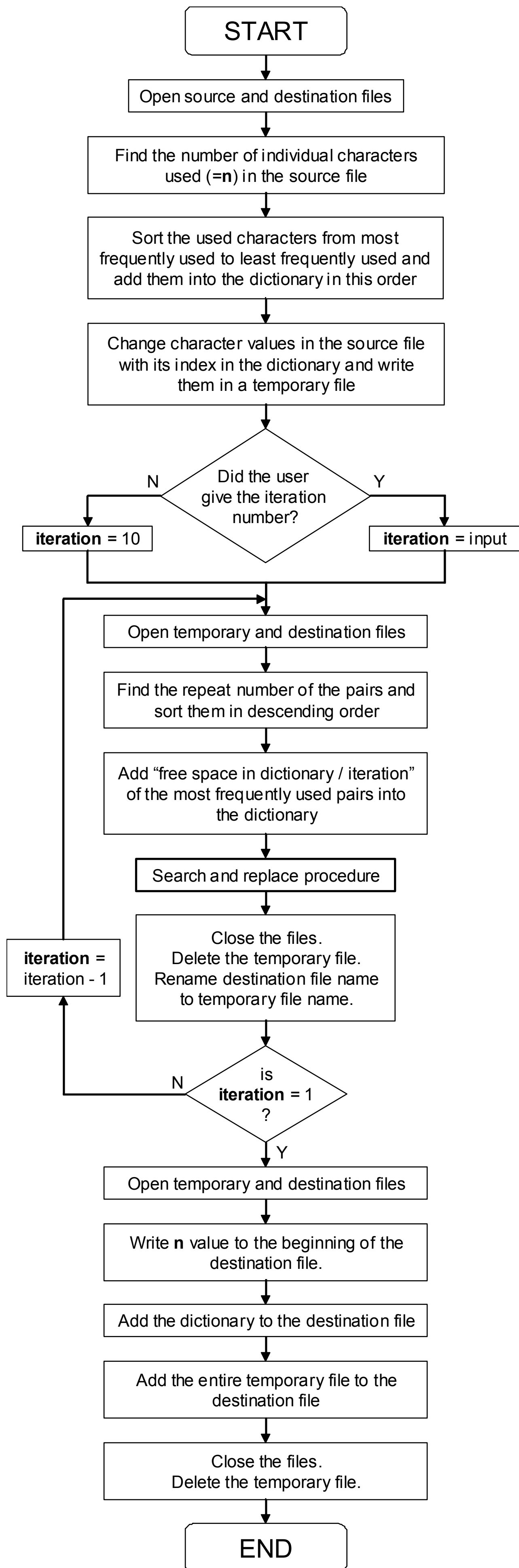

Figure 2. Flowchart of RSSDC

```
Reverse_Search_And_Replace(int source, file dest){
    if (source < n){
        - source is an individual character -
        write the dictionary meaning of
        the source to dest
    } else {
        - source is a digram -
        Reverse_Search_And_Replace (
               1st character of the source, dest);
        Reverse_Search_And_Replace (
               2nd character of the source, dest);
    }
}
```

Like SSDC decoder, RSSDC decoder is also a one-pass coder and it works very fast.

PERFORMANCE OF THE ALGORITMS

In Table 1, we give the results of compressing the fourteen commonly used files of the Calgary Compression Corpus [7] with our algorithms. In this table, **ds** represents dictionary size and **i** represents total number of iterations used in RSSDC. Compression efficiency is expressed as output bits per input character. The Compression time measurements were made on a computer which has an Intel Pentium4 1.7 GHz processor and 256MB of total RAM. CPU time is given rather than elapsed time so the time spent performing I/O is excluded.

The c codes of the other algorithms that are used in this comparison are; for Huffman Coding “codhuff.c” and for LZW “codlzw.c” both by David Bourgin (1995), for Arithmetic Coding “ari.cpp” by Mark Nelson (1996) and for LZ77 “prog1.c” by Rich Geldreich, Jr. (1993).

	Compressed Size (bytes)	bits/ char	Comp. Time (s)	Decomp. Time (s)
Uncompressed	3.141.622	8,00		
HUFFMAN	1.764.418	4,49	0.64	0.45
ARITHMETIC	1.713.128	4,36	1.09	1.19
LZW	1.521.341	3,87	0.59	0.34
LZ77	1.347.216	3,43	0.88	0.13
SSDC, ds=256	2.003.492	5,10	1.00	0.14
SSDC, ds=512	1.936.050	4,93	1.58	0.28
RSSDC, ds=256, i=5	1.736.729	4,42	4.22	0.16
RSSDC, ds=256, i=10	1.715.404	4,37	6.73	0.16
RSSDC, ds=256, i=15	1.709.426	4,35	9.33	0.16
RSSDC, ds=256, i=20	1.708.191	4,35	11.58	0.16
RSSDC, ds=512, i=5	1.493.825	3,80	9.61	0.30
RSSDC, ds=512, i=10	1.461.175	3,72	14.16	0.30
RSSDC, ds=512, i=15	1.453.582	3,70	18.64	0.30
RSSDC, ds=512, i=20	1.444.759	3,68	24.47	0.30

Table 1. Results of compressing Calgary Corpus

The table shows that compression improves with increasing dictionary size and total number of iterations. However, while the compression ratio increases, the compression time also increases.

Like other dictionary-based algorithms, the decompression speed of our algorithms is faster than the compression speed. It is clearly seen that, the decompression time does not depend on to the total number of iterations in the compression. Because, no matter how much iteration used in the compression, the decompression is always done in one-pass.

When we look at the SSDC results, we can see that the compression efficiency of non-recursive approach is not good, but the algorithm runs fast. SSDC cannot compress more than 50% by the nature of its structure. For example, the “pic” file is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros. However, SSDC cannot compress the “pic” file of the Calgary Corpus more than 50% (see Appendix). It is obvious that, if trigrams used instead of digrams in SSDC, compression ratio will be increased.

Because of all the ASCII characters are used in geo, obj1 and obj2 files, if we use the dictionary size 256, the algorithm cannot find a free space to fill. Thus, the use of 256 causes an expansion instead of compression (see Appendix).

CONCLUSION

The approach presented in this paper can be used when fast decompression is necessary. For example, it can be used to prepare a setup for software. The software files are compressed once when the setup is prepared, but later, the decompression is made many times when installation of the software. Therefore, in this kind of situations, the decompression speed is more important than the compression speed, and this is because the decompression speed of RSSDC algorithm is valuable.

By making the following improvements to these algorithms, the compression time can be decreased and the compression ratio can be increased.

- Compression speed of these algorithms can be decreased by using more efficient searching and sorting algorithms.
- Compression ratio can be increased by using trigrams or tetragrams instead of digrams. However, in this situation, compression time will be decreased.
- Elimination of unnecessary items from the dictionary may increase compression ratio, but it may also increase compression time. For example, if the algorithm compresses “_the” with “_t” + “he”, the pair in the middle “th” can be removed if it is not used many times in other character groups.
- A part of the dictionary can be made static. For example, 96 printable characters and 32 most frequently used pairs in English language can be placed in 0-127 interval of the dictionary. Later, the first-pass of our algorithm will be filling the rest of the dictionary. By doing this improvement compression time might decrease, but compression ratio also decreases for some source files, which include digrams that do not match static part of the dictionary.

REFERENCES

[1] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of IRE, 40(9):1098-1101, September 1952.

[2] D. E. Knuth. Dynamic Huffman Coding. J. Algorithms, 6(2):163-180, June 1985.

[3] A. Moffat, R.M. Neal, I.H. Witten. Arithmetic Coding Revisited. ACM Transactions on Information Systems, 16:256-294, 1995.

[4] K. Sayood. Introduction to Data Compression. San Francisco, California, Morgan Kaufmann, 1996.

[5] T. A. Welch. A Technique for High-Performance Data Compression. IEEE Computer, 17(6):8-19, June 1984.

[6] I. H. Witten, R.M. Neal, R.J. Cleary. Arithmetic Coding for Data Compression. Communications of the ACM, 30:520-540, 1987.

[7] I. H. Witten and T. Bell. The Calgary/Canterbury text compression corpus. Anonymous ftp from ftp.cpsc.ucalgary.ca: /pub/text.compression.corpus/

[8] J. Ziv, A. Lempel. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, IT-23(3):337-343, May 1977.

[9] J. Ziv, A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory, IT-24(5):530-536, September 1978.

APPENDIX

File Name	Uncompressed size (byte)	SSDC ds=256	SSDC ds=512	RSSDC ds=256 i=5	RSSDC ds=256 i=10	RSSDC ds=256 i=15	RSSDC ds=256 i=20	RSSDC ds=512 i=5	RSSDC ds=512 i=10	RSSDC ds=512 i=15	RSSDC ds=512 i=20
bib	111.261	67.461	67.142	58.458	56.380	56.241	56.012	53.953	51.549	50.411	50.469
book1	768.771	439.511	444.416	420.813	414.990	413.792	414.530	390.308	384.352	383.979	382.582
book2	610.856	367.554	368.193	354.952	353.072	350.823	350.982	323.564	321.901	322.910	321.220
geo	102.400	102.657	75.696	102.657	102.657	102.657	102.657	64.991	63.631	63.295	63.174
news	377.109	248.074	242.665	242.860	240.882	241.033	240.358	226.582	222.835	221.055	218.376
obj1	21.504	21.761	17.205	21.761	21.761	21.761	21.761	13.993	13.600	13.489	13.407
obj2	246.814	247.071	184.515	247.071	247.071	247.071	247.071	154.122	150.938	149.973	149.614
paper1	53.161	32.838	33.101	31.193	30.811	30.761	30.679	28.298	27.548	27.504	27.312
paper2	82.199	47.411	48.671	45.023	44.677	44.211	44.176	40.984	40.151	40.141	39.890
pic	513.216	271.103	296.006	78.145	71.493	70.328	69.731	74.715	66.764	65.644	65.304
progc	39.611	25.215	25.410	23.020	22.184	22.063	22.008	20.871	20.407	20.080	19.720
progl	71.646	41.959	42.608	35.765	34.758	34.589	34.630	32.155	30.759	30.110	29.512
progp	49.379	30.190	30.523	24.546	23.644	23.163	23.088	21.564	20.436	19.573	19.366
trans	93.695	60.687	59.899	52.443	51.024	50.933	50.508	47.056	45.635	44.750	44.141
Total	**3.141.622**	**2.003.492**	**1.936.050**	**1.738.707**	**1.715.404**	**1.709.426**	**1.708.191**	**1.493.156**	**1.460.506**	**1.452.914**	**1.444.087**
Compression Time		**1.00s**	**1.58s**	**4.22s**	**6.73s**	**9.33s**	**11.58s**	**9.61s**	**14.16s**	**18.64s**	**24.47s**
Decompression Time		**0.14s**	**0.28s**	**0.16s**	**0.16s**	**0.16s**	**0.16s**	**0.30s**	**0.30s**	**0.30s**	**0.30s**

Detailed results of compressing Calgary Corpus with SSDC and RSSDC

File Name	Uncompressed size (byte)	Huffman Coding	Arithmetic Coding	LZW	LZ77
bib	111.261	72.941	72.789	60.307	47.070
book1	768.771	438.577	436.883	419.111	393.448
book2	610.856	368.521	364.720	325.274	263.106
geo	102.400	73.084	72.400	79.287	83.544
news	377.109	246.606	244.471	231.578	183.705
obj1	21.504	16.584	16.038	13.684	12.155
obj2	246.814	194.635	187.294	134.562	101.840
paper1	53.161	33.550	33.120	28.960	22.776
paper2	82.199	47.830	47.535	42.705	37.073
pic	513.216	106.948	74.804	65.842	118.912
progc	39.611	26.111	25.920	21.143	16.494
progl	71.646	43.181	42.619	30.499	21.978
progp	49.379	30.416	30.209	21.502	15.137
trans	93.695	65.434	64.326	46.887	29.978
TOTAL	**3.141.622**	**1.764.418**	**1.713.128**	**1.521.341**	**1.347.216**
Compression Time		**0.64s**	**1.09s**	**0.59s**	**0.88s**
Decompression Time		**0.45s**	**1.19s**	**0.34s**	**0.13s**

Detailed results of compressing Calgary Corpus with Huffman Coding, Arithmetic Coding, LZW and LZ77