Computing and Informatics, Vol. 29, 2010, 741–756

ISSDC: DIGRAM CODING BASED LOSSLESS DATA COMPRESSION ALGORITHM

Altan Mesut, Aydin Carus

Computer Engineering Department
Trakya University
Ahmet Karadeniz Yerleskesi
Edirne, Turkey
e-mail: {altanmesut, aydinc}@trakya.edu.tr

Manuscript received 18 August 2008; revised 14 January 2009
Communicated by Jacek Kitowski

Abstract. In this paper, a new lossless data compression method that is based on digram coding is introduced. This data compression method uses semi-static dictionaries: All of the used characters and most frequently used two character blocks (digrams) in the source are found and inserted into a dictionary in the first pass, compression is performed in the second pass. This two-pass structure is repeated several times and in every iteration particular number of elements is inserted in the dictionary until the dictionary is filled. This algorithm (ISSDC: Iterative Semi-Static Digram Coding) also includes some mechanisms that can decide about total number of iterations and dictionary size whenever these values are not given by the user. Our experiments show that ISSDC is better than LZW/GIF and BPE in compression ratio. It is worse than DEFLATE in compression of text and binary data, but better than PNG (which uses DEFLATE compression) in lossless compression of simple images.

Keywords: Lossless data compression, dictionary-based compression, semi-static dictionary, digram coding

1 INTRODUCTION

Lossy and lossless data compression techniques are widely used to increase capacity of data storage devices and to transfer data faster on networks. Lossy data compression reduces the size of the source data by permanently eliminating redundant information. When the file is uncompressed, only a part of the original information is retrieved. Lossy data compression techniques are generally used for photo, video and audio compression, 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. Typical examples are represented by text documents, executable programs and source codes. Lossless compression techniques are generally classified into two groups: entropy based coding and dictionary based coding.

In entropy based coding, compression takes place based on the frequency of input characters or character groups. The symbols that occur more frequently are coded with shorter codewords. For this reason this method is also known as variable length coding (VLC). A VLC technique can be combined with other compression techniques to improve compression ratio and it is generally used at the end of the compression process. The best known entropy based techniques are Huffman Coding [8, 10] and Arithmetic Coding [11, 20].

In dictionary based coding, frequently used symbol groups (characters, pixels or any other type of binary data) are replaced with related indexes of a dictionary. Dictionary-based techniques can be divided into three categories. In static dictionary scheme, the dictionary is the same for all inputs. In semi-static dictionary scheme, distributions of the symbols in the input sequence are learned in the first pass, compression of the data is made in the second pass by using a dictionary derived from the distribution learned. In adaptive (dynamic) dictionary scheme, the dictionary is built on the fly (or it needs not to be built at all, it exists only implicitly) using the source seen so far. Static dictionary scheme is the fastest in compression time, but it is only 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. In semi-static dictionary scheme, the dictionary must be sent as side information with the compressed source. For this reason, it is not suitable for small files. Adaptive techniques are generally faster than semi-static techniques; because of their capability of doing all of the jobs in one pass.

Most adaptive dictionary-based techniques have their roots in two landmark papers by Jacob Ziv and Abraham Lempel. The first paper has been published in 1977 [21], and for this reason the algorithm that is presented in this paper is known as LZ77. The LZ77 algorithm works by keeping a history window of the most recently seen data and comparing the current data being encoded with the data in the history window. What is actually placed into the compressed stream are references to the position in the history window, and the length of the match. A slightly modified version of LZ77 that provides better compression ratio is described by Storer and Szymanski [17] in 1982 (LZSS). LZ77 family algorithms are generally combined with a VLC algorithm to improve compression ratio. For example DEFLATE algorithm [6], which is used in Gzip data compressor and PNG image file format, is a combination of the LZSS Algorithm and Huffman Coding. The second paper by

Lempel and Ziv has been published in 1978 [22] and the algorithm in this paper (LZ78) works by entering phrases into a dictionary and then, when a reoccurrence of that particular phrase is found, outputting the dictionary index instead of the phrase. There are many modifications of LZ78 algorithm and the most well known one is Terry Welch's LZW Algorithm [19]. LZW is more commonly used to compress binary data, such as bitmaps. UNIX compress and GIF image compression format are both based on LZW.

Digram coding is one of the best known static dictionary encoder that has been proposed several times in different forms [3, 4, 9, 15, 16, 18]. In this paper, static and semi-static forms of digram coding are explained in Section 2. Our algorithm, which improves the compression ratio of semi-static digram coding by repeating the encoding process several times, is described in Section 3. Experimental results and the evaluation of our algorithm are given in Section 4 and Conclusion is given in Section 5.

2 DIGRAM CODING

Digram coding is a static dictionary technique that is less specific to source data. 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 [14].

The Digram Encoder reads a pair from the source and searches the dictionary to see if this pair exists in the dictionary. If it does, the corresponding index is encoded and the encoder reads another pair from the source for the next step. 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 and the encoder reads another character to complete this digram. This *search and replace procedure* is repeated until the end of the source. The algorithm of the digram encoder is given in Figure 1.

```
Chr1 = Read a character from the source
Do until end of source {
    Chr2 = Read a character from the source
    If the digram exist in dictionary {
          The corresponding dictionary index is encoded
          Chr2 = Read a character from the source
    }
    Else {
          Chr1 is encoded
    }
    Chr1 = Chr2
}
```

Fig. 1. The algorithm of the digram encoder

A semi-static implementation of digram coding should contain a two-pass mechanism. In the first pass, all of the individual characters and digrams that are used in the source are found. All of the individual characters are added to the first part of the dictionary and the most frequently used digrams are added to the second part of the dictionary. If the source contains n individual characters, and the dictionary size is d, then the number of digrams that can be added to the dictionary is $d - n$. The decoder must know this n value to determine the length of the first and the second parts of the dictionary. The n and d values and the dictionary that contains n individual characters and $d - n$ digrams should be written in the destination file (or sent to receiver) as side information. The size of the dictionary is given in Equation (1). The second pass of the compression process is similar to the static digram coding.

$$\text{dictionary size} = 2 + [n + 2(d - n)] = 2d - n + 2\,\text{bytes} \tag{1}$$

The decompression process is also similar in both static and semi-static implementations. The semi-static implementation uses a one-pass decoding algorithm like the static one. The main difference between them is that the semi-static implementation obtains the dictionary (with the n and d values) before the decoding while the static one has it already. After the dictionary is obtained, all of the elements of the compressed data are changed with their dictionary meanings in the decompression process. This simple decoding algorithm runs much faster than the encoding algorithm.

BPE (Byte Pair Encoding), which is developed by Philip Gage [7], is a multi-pass digram coding algorithm. The algorithm compresses data by finding the most frequently occurring pairs of adjacent bytes in the data and replacing all instances of the pair with a byte that was not in the original data. This process is repeated until no further compression is possible, either because there are no more frequently occurring pairs or there are no more unused bytes to represent pairs. The table of pair substitutions is written before the data is packed. The idea behind ISSDC is similar with the idea of BPE. However, the encoding process of ISSDC is entirely different from BPE (Section 3.3) and ISSDC is able to compress more than BPE (Table 3).

3 ITERATIVE SEMI-STATIC DIGRAM CODING (ISSDC)

We developed an algorithm that is based on semi-static digram coding and we used an iterative approach in this algorithm to improve the compression ratio. This multi-pass algorithm does not fill all of the second part of the dictionary in one pass. The second part is divided by the number of iterations and each iteration fills its own free space. A digram which is added in the n^{th} iteration will become a character in the $(n+1)^{\text{th}}$ iteration.

In most English texts, the 'e' character pair (e and space) is the most frequently occurring digram. For example, *book2* file of the Calgary Compression Corpus [1, 2], which is 610 856 bytes in size, contains 15 219 'e' pairs. This means that if we can encode only this pair with 1 byte instead of 2 bytes, the file size will be decreased to 595 637 bytes. The 10 most frequently occuring pairs in this file are given in Table 1.

Pair		ASCII		Occurrence Number
e		101	32	15 219
	t	32	116	11 205
t	h	116	104	10 226
s		115	32	9 591
h	e	104	101	8 854
	a	32	97	8 391
i	n	105	110	7 779
	s	32	115	7 341
e	r	101	114	7 030
o	n	111	110	6 837
Total				92 473

Table 1. The 10 most frequently occuring pairs in book2 file

If the 10 most frequently occuring pairs in Table 1 are added to a dictionary and digram coding is performed with this dictionary to compress *book2* file, some words can be encoded in different ways. For example, if the space character before 'the' word was compressed together with the character before it, then the 'the' word will be compressed with 'th' and 'e' digrams. Otherwise, it will be compressed with 't' and 'he' digrams. This means that the compression gains of these digrams will be less than their occurrence numbers and the decrease in file size will not be 92 473 bytes (it will be 72 960 bytes).

ISSDC algorithm eliminates some digrams to avoid this inconsistency. A digram is eliminated if its first character is equal to the second character – or its second character is equal to the first character - of one of the digrams that are already added in the current iteration. Therefore, if we use ISSDC algorithm, the 't', 'he', 'a' and 's' digrams in Table 1 will not be added to the dictionary in the first iteration.

The *book2* file contains 96 individual characters and there are 160 codes left for representing digrams. If the iteration number parameter of ISSDC is set as 16, every iteration adds 10 most frequently occuring pairs (except eliminated ones) to the dictionary. The pairs that are added to the dictionary in the first iteration are given in Table 2. In this table, dictionary indexes start at index 96, because the 0–95 interval is used for representing the individual characters of the source. After the digram coding is performed with this dictionary on *book2* file, the file size is decreased by 76 762 bytes and the new file contains 106 individual characters. If the '97,96' digram is added to the dictionary in the next iteration, the 'the' word and the space character after it can be encoded as a number between 106 and 115 (4 characters compressed to 1 byte).

Dictionary Index	Pair		ASCII		Occurrence Number
96	e		101	32	15 219
97	t	h	116	104	10 226
98	s		115	32	9 591
99	i	n	105	110	7 779
100	e	r	101	114	7 030
101	o	n	111	110	6 837
102	t		116	32	6 238
103	o	r	111	114	4 772
104	e	n	101	110	4 583
105	LF	.	10	46	4 487
	Total				76 762

Table 2. The pairs that are added to the dictionary in the first iteration

3.1 Compression Algorithm

The compression algorithm copies the source file into RAM to avoid large amount of file I/O operations. During this copy process, the characters that are used by the source are also found. These characters and the total number of them (n) are stored to form the first part of the dictionary. The real codes of the characters (ASCII codes) in RAM are changed with dictionary indexes of these characters.

If the number of iterations (i) and the length of the dictionary (d) are given, the second part of the dictionary is divided into i equal parts. Thus, in every iteration $(d-n)/i$ digrams are added to the second part of the dictionary. After that, the copy of the source file in RAM is compressed by the digram encoder.

ISSDC algorithm also contains mechanisms that can decide about the number of iterations and the length of the dictionary automatically when they are not given. If the number of iterations is given, but the dictionary size is not, dictionary size is accepted as 1 024, and the algorithm works similarly as explained above. If the dictionary size is given, but the number of iterations is not, each iteration step continues until the repeat number of the most frequently used digram that is found in current iteration is halved. This means that, in each iteration step, the repeat number of the digrams that are added to the dictionary must be larger than or equal to half that of the first added one. So, the iteration number and the number of digrams that are added in each iteration are not known at the beginning.

If both the number of iterations and the dictionary size are not given, the algorithm runs in *automatic decision mode*. The initial dictionary size will be the smallest integer, which is power of two and larger than 2^n. After the initial dictionary size is defined, the method that adds the digrams to the dictionary until most frequently occurring digram is halved is used; but this time, when the dictionary is completely filled, a check is made for deciding whether doubling the dictionary size is necessary or not. Doubling the dictionary size means that one more bit must be used to represent each character. For example, if we increase the dictionary size from 256 to 512, we must use 9 bits to represent a character, and this will increase the source size by 12.5 % (percentage of $9/8 - 1$). Doubling the dictionary size will not be effective when the compression ratio cannot cover up this expense.

It is hard to predict whether it is valuable or not, before doubling the dictionary and making the compression. ISSDC uses the repeat number of the digram that is most frequently occurred in last iteration (m) for making a decision. A threshold is found with dividing this number by 2 and multiplying by dictionary size. If 12.5 % of the source is smaller than this threshold, the dictionary size is doubled. This threshold value is defined with the help of many compression test results.

Assume that the repeat number of the most frequently occurring digram is $m = 20$. If the dictionary is doubled when $d = 256$, there will be a free space in dictionary for 256 new digrams. We can predict that these 256 digrams have an average repeat number of $m/2 = 10$. When we change 10 digrams with 10 characters, the size of the source is decreased by 10 characters. Therefore, we can say that the size of the source can be decreased by $256 \times 10 = 2\,560$ characters (2 560 bytes) on average. If 12.5 % (1/8) of the source is smaller than 2 560, ISSDC predicts that doubling the dictionary size can be effective.

It can be easily calculated that, if the dictionary size is increased from 512 to 1 024, the expense in the source size will be 11.1 % (percentage of $10/9 - 1$), and if it is increased from 128 to 256, the expense will be 14.3 % (percentage of $8/7 - 1$). Although these values are a little far from 12.5 %, ISSDC use 12.5 % for all conditions, because this little difference will not affect the success of the prediction in a large amount. Therefore, the main criterion about increasing the dictionary size is given in Equation (2).

$$d \times m \div 2 < \text{File Size} \div 8 \tag{2}$$

If the m value is too small, even if the criterion given above is true, if the dictionary size is increased, repeat numbers of digrams might decrease to zero and cause infinite loop. Therefore, we need an extra criterion to avoid this infinite loop state. We select this criterion as follows: if the m value is smaller than 8, do not increase the dictionary size. Another extra criterion is “do not increase the dictionary size if it is 1 024”. The reason is that when $d = 2\,048$ compression ratio will not be changed in a large amount and compression time will be increased. The final criteria are given in Equation (3).

$$d \times m < \text{File Size} \div 4 \text{ and } d < 1\,024 \text{ and } m > 8 \tag{3}$$

After all iterations are finished and the dictionary is completely filled, the n and d values and the dictionary are added to the beginning of the destination file. ISSDC uses five different dictionary sizes, namely 64, 128, 256, 512 and 1 024. In order to represent d in 1 byte (256, 512 and 1 024 cannot be represented in 8 bits), $\log_2 d$ is used instead of the d value. After the dictionary, the compressed state of the source file is added and the destination file is closed. Figure 2 shows the parts of the destination file.

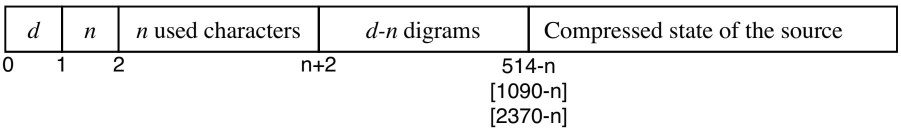

Fig. 2. Parts of the destination file

If d is greater than 256, the size of the digrams part of the dictionary in the destination file is not $2(d-n)$ bytes like given in Equation (1). Because the values of the digrams in 257–512 interval may be larger than 255 and the values of the digrams in 513–1 024 interval may be larger than 511, these items must be represented with 9 bits and 10 bits, respectively. The size of the dictionary in destination file when d is equal to 1 024 is given in Equation (4).

$$2 + n + 2(256 - n) + 2(256) \times 9/8 + 2(512) \times 10/8 = 2\,370 - n\text{bytes} \qquad (4)$$

If d is 256, ISSDC uses the standard *fputc* C function for all parts of the destination file; but if d is not 256, it uses some other special functions in order to write smaller 6 bits and 7 bits values and larger 9 bits and 10 bits values. These functions affected the speed of the algorithm in a negative way.

The dictionary size and the number of iterations can be given as parameters to ISSDC encoder. If these values are not given as parameters, it is assumed that they are zero, and the algorithm runs in automatic decision mode. ISSDC algorithm is given in Figure 3.

An example is given below to clarify the compression process.

Assume to compress *abracadabra* word with ISSDC. In the first pass 5 characters that are used in this word (a, b, c, d and r) are found and they are added to the dictionary. In the second pass (the first pass of the first iteration), the most frequently used digrams, which are ab, br and ra are found, and they are added to 5^{th}, 6^{th} and 7^{th} places in the dictionary. In the second pass of the first iteration, the digram encoder compressed the word using the dictionary. In second iteration, the same process is done for $5(ab), 7(ra)$ digram. Figure 4 illustrates the compression process.

3.2 Decompression Algorithm

ISSDC decoder is a one-pass coder and it works very fast. It uses a recursive approach for obtaining the extraction of a digram quickly. Think about decoding $8(abra)$ that is given in the previous example. In the first iteration 8 is decompressed as 5 and 7. These two characters represent digrams, because they are not smaller than the n value, which is 5. Thus, recursive process must continue for both of

```
I = number of iterations (given as a parameter)
D = dictionary size (given as a parameter)
Open source and destination files
Find used characters while copying source file to RAM
N = number of used characters
Add used characters to the first part of the dictionary
While D > N {
   If I and D are given { limit = N + (D - N) / I }
   If I is given but D is not { limit = N + (1024 - N) / I }
   If D is given but I is not { limit = D }
   If I and D are not given{
      D = limit = the smallest integer which is
                        power of 2 & greater than 2 * N
   }

   Find digrams and sort them in descending order
                        according to # of their occurrance

   M = Occurrence # of the most freq. occurred digram

   If I is given {
      Add (limit – N) most freq. occurred digrams to dictionary
   } Else {
      Add digrams to dict. while Repeat # of digrams >= M/2
   }

   If D is not given & D < 1024 & D*M > SourceSize/4 & M > 8 {
      D = D * 2
   }

   Perform Digram Coding (source is compressed in RAM)

   N = limit
   If I > 0 { I = I – 1 }
}
Write D, N, dictionary & the source in RAM to destination file
Close source and destination files
```

Fig. 3. The compression algorithm of ISSDC

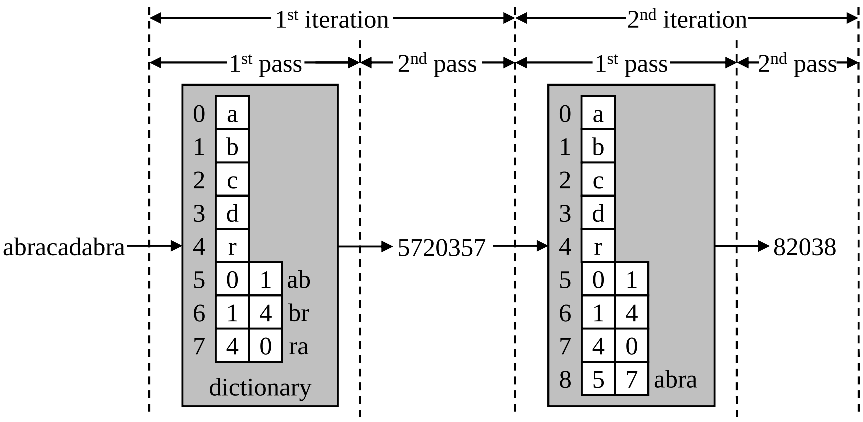

Fig. 4. Compression process with ISSDC

them. In the second iteration, 5 is decompressed as 0 and 1, and 7 is decompressed as 4 and 0. All of these values are smaller than 5. This means that they represent characters, not digrams. As a result, $0140(abra)$ is extracted. The recursive function of Digram Decoding used in ISSDC is given in Figure 5.

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

Fig. 5. The recursive digram decoding procedure used in ISSDC

3.3 Similarities and Differences Between ISSDC and BPE

Both algorithms perform multi-pass digram coding and both of them use the unused ASCII codes to represent digrams. However, while BPE cannot change the dictionary size, ISSDC is able to increase the dictionary size to 512 or 1 024, and decrease it to 128 or 64 for better compression ratio. There is no need to give any parameters to ISSDC, it can be run in *automatic decision* mode. For example, if the source file is too small and there are very few different symbols, there will be not much *frequently occurring pairs* in the source. In this kind of situation, ISSDC automatically chooses 64 for dictionary size. This means that every symbol in the source will be represented with 6 bits instead of 8 bits and so the size of the source will be decrased by 25 % even if there are no *frequently occurring pairs* to compress.

ISSDC compresses the whole source in one loop, while BPE divides the source into blocks and compresses each block separately. In every iteration of BPE, the algorithm finds the most frequently occurring pair and replaces the pair throughout the data buffer with an unused character. However, ISSDC algorithm can handle more than one frequently occurring pair in the same iteration. Therefore, the number of iterations in ISSDC is less than the number of iterations in BPE, but the iteration procedure of ISSDC is larger and slower.

Both algorithms perform expansion in a single-pass mechanism. The difference between them is the expansion of ISSDC uses a recursive function, while the expansion of BPE includes a stack structure. However, this is not a big difference since recursive functions use stacks implicitly.

4 EXPERIMENTAL RESULTS

We made two different comparisons to evaluate the performance of ISSDC: The first one is about the performance of ISSDC on different data types while the second one is only concerned with the performance of ISSDC on images.

4.1 Evaluation Methodology

In our first comparison, we compared ISSDC with another digram coding based algorithm (BPE), a traditional dictionary-based algorithm (LZW) and one of the best dictionary-based algorithms that uses LZSS and Huffman Coding (DEFLATE). We used Gzip 1.2.4 for DEFLATE, the C code of Mark Nelson for LZW [12] and the C code of Philip Gage for BPE [7]. Both of these codes were compiled with GCC compiler with Best Optimization Option (-O3). All algorithms were used with their maximum compression options: -9 for DEFLATE; BITS = 14 for LZW; BLOCKSIZE = 10 000, HASHSIZE = 8 192, MAXCHARS = 200 and THRESHOLD = 3 for BPE. *The Calgary Compression Corpus* [1, 2] was used as test data for this comparison. This corpus contains 14 files that are 3 141 622 bytes in size.

In our second comparison, we chose GIF [5] and PNG [13] methods as references. Both of these widely used lossless image compression methods use dictionary-based data compression algorithms like ISSDC (GIF uses LZW algorithm and PNG uses DEFLATE algorithm). For this comparison, we selected 11 organisation logos[1] from different internet sites that have a small number of colors and low complexity. We did not select photographs, because photographs are generally compressed with lossy techniques. NConvert v4.95 image compression utility is used to perform GIF and PNG compression. In PNG compression *-clevel 9* parameter is used to obtain maximum compression ratio.

All of these logos were converted first to 8-bit per pixel PCX image format and then compressed with GIF, PNG and ISSDC algorithms. We choose PCX instead of BMP because it has smaller overhead, which equals 896 bytes (128 (header) + 768 (map table: 3×256) = 896). The overhead of BMP is generally 1 078 bytes (54 (header) + 1 024 (map table: 4×256) = 1 078), but if the image width is not multiple of 4, it will be larger. ISSDC compresses file header and map table together with pixel data.

The time measurements in both comparisons were evaluated on a computer that has Intel Core 2 Duo T5500 1.66 GHz CPU and 2 GB 667 MHz DDR2 RAM.

[1] `http://conferences.computer.org/icws/2005/images/IEEE-logo.gif`,
`http://www.ilofip.org/pictures/ilo\textunderscorelogo.gif`,
`http://blogs.zdnet.com/open-source/images/iso\textunderscorelogo.gif`,
`http://www.fao.org/sd/2002/img/KN0801fao.gif`,
`http://www.dalequedale.com/media/LogoUnicef.gif`,
`http://www.worldbank.org/wbi/qcs-1/logos/Logo-WBank.gif`,
`http://www.baumholder.army.mil/media/det7/natologo.gif`,
`http://www2.oecd.org/pwv3/NewLogoOECD.GIF`,
`http://www.worldatlas.com/webimage/flags/specalty/olympic.gif`,
`http://www.ioccg.org/news/Feb2008/unesco.gif`,
`http://www.leb.emro.who.int/search/whologofinal7.bmp`.

4.2 Results of Comparisons

The results of compressing *The Calgary Compression Corpus* with BPE, LZW and DEFLATE algorithms with their best compression ratio modes and ISSDC with its automatic decision mode are presented in Table 3. Table 4 has four different ISSDC results which are different from each other with their parameter combinations. In both of these tables bold values indicate the best results and compression efficiency is expressed as output bits per input character.

File Name	File Size	ISSDC	BPE	LZW	DEFLATE
bib	111 261	43 294	56 631	48 641	**34 900**
book1	768 771	337 491	415 094	348 412	**312 281**
book2	610 856	276 044	321 566	293 823	**206 158**
geo	102 400	**62 725**	73 707	79 520	68 414
news	377 109	195 837	231 976	201 643	**144 400**
obj1	21 504	12 722	13 313	15 872	**10 320**
obj2	246 814	130 155	149 892	208 944	**81 087**
paper1	53 161	23 482	28 529	26 901	**18 543**
paper2	82 199	33 760	42 148	38 503	**29 667**
pic	513 216	61 096	61 833	65 656	**52 381**
progc	39 611	17 096	20 506	20 966	**13 261**
progl	71 646	24 494	29 734	28 939	**16 164**
progp	49 379	16 132	20 872	21 033	**11 186**
trans	93 695	36 317	44 869	41 777	**18 862**
Total Size (bytes)		1 270 645	1 510 670	1 440 630	**1 017 624**
Efficiency (bits/char)		3.24	3.85	3.67	2.59
Compression Time (s)		1.56	1.87	0.35	0.75
Decompression Time (s)		0.16	0.32	0.37	0.46

Table 3. Results of compressing Calgary Corpus

Table 3 shows that ISSDC automatic decision mode is worse than DEFLATE, but better than the other two algorithms in compression efficiency. In all files of Calgary Corpus, ISSDC has the second best compression ratio except it has the best ratio in geo file. Although ISSDC is the best algorithm in decompression speed, it is only better than BPE in compression speed.

Table 4 shows that compression ratio improves with increasing dictionary size and total number of iterations. However, the efficiency of increasing the number of iterations is lowered at a particular point, and after that point the compression ratio improves a little while the compression time increases linearly. It can be seen from Table 3 and Table 4 that automatic decision mode can obtain an optimal solution that has a good compression ratio with an acceptable compression time.

Like with other dictionary-based algorithms, the decompression speed of ISSDC 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,

File Name	File Size	$d = 512$ $i = 10$	$d = 512$ $i = 20$	$d = 1\,024$ $i = 10$	$d = 1\,024$ $i = 20$
bib	111 261	48 805	47 865	**43 167**	43 355
book1	768 771	366 843	365 074	344 053	**339 278**
book2	610 856	306 686	308 422	280 097	**275 314**
geo	102 400	62 807	**62 717**	64 539	64 167
news	377 109	215 630	212 358	198 217	**196 074**
obj1	21 504	13 048	12 996	12 757	**12 732**
obj2	246 814	148 710	148 082	131 300	**130 461**
paper1	53 161	26 201	25 927	24 171	**23 402**
paper2	82 199	37 452	37 039	34 539	**33 854**
pic	513 216	63 773	62 942	63 363	**61 237**
progc	39 611	19 191	19 015	17 539	**17 243**
progl	71 646	28 910	29 157	25 266	**24 393**
progp	49 379	18 901	18 420	16 867	**16 206**
trans	93 695	42 866	42 538	37 089	**35 938**
Total Size (bytes)		1 399 823	1 392 552	1 292 964	**1 273 654**
Efficiency (bits/char)		3.56	3.55	3.29	3.24
Compression Time (s)		1.09	1.33	1.39	1.88
Decompression Time (s)		0.16	0.16	0.16	0.16

Table 4. Results of ISSDC with different parameter combinations

no matter how many iterations are used in the compression, the decompression is always done in one-pass.

The results of compressing selected images are given in Table 5. In this table bold values indicate the best results. In this comparison ISSDC was used in automatic decision mode while PNG was used in its best compression ratio mode and GIF was used with 89a format. PNG gives the best results only in 2 files while ISSDC is the best in 9 files. The results show that the compression ratio of ISSDC is very good when it is used with simple images.

5 CONCLUSION

The dictionary-based compression method presented in this paper can achieve good compression ratio especially with simple images and good decompression speed for all types of data. In most cases, the decompression speed is more important than the compression speed because it is more often used (images are coded once but viewed many times, files of an application are compressed once when the setup of this application is prepared, but later, decompression is performed many times for installation of this software, etc.). For this reason, the decompression speed of ISSDC algorithm is valuable.

Elimination of unnecessary items from the dictionary may increase compression ratio, but it may also increase compression time. For instance, in the example of Section 3.1, ISSDC compressed *abra* with *ab* and *ra* digrams. Therefore *br* digram

Image Name	width	height	ISSDC	PNG	GIF
IEEE-logo	458	147	**2 395**	3 233	4 286
ilo_logo	400	377	**7 702**	9 031	9 775
iso_logo	223	205	**3 336**	3 997	4 609
KN0801fao	473	473	**21 349**	23 583	23 784
LogoUnicef	295	269	23 128	**22 545**	25 691
Logo-WBank	510	224	**8 436**	9 155	11 687
natologo	500	375	**9 324**	10 462	11 077
NewLogoOECD	228	63	**2 293**	2 799	3 078
olympic	316	209	**5 971**	6 130	6 495
unesco	373	292	**2 712**	3 401	6 187
who logo final 7	383	312	14 706	**14 507**	16 656
Total Size (bytes)			**101 352**	108 843	123 325
Compression Time (s)			0.30	0.87	0.21

Table 5. Lossless image compression results

in the 6$^{\text{th}}$ place of the dictionary is never used. We developed a mechanism that can eliminate unused digrams from the dictionary. By using this mechanism the compression ratio was increased by nearly 3 %. However, this implementation was not very suitable because of its negative effect to the compression time.

REFERENCES

[1] Bell, T. C.—Cleary, J. G.—Witten, I. H.: Text Compression. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] Bell, T. C.—Witten, I. H.—Cleary, J. G.: Modeling for Text Compression. Computing Surveys, Vol. 21, 1989, No. 4, pp. 557–591.

[3] Bookstein, A.—Fouty, G.: A Mathematical Model for Estimating the Effectiveness of Bigram Coding. Information Processing and Management, Vol. 12, 1976, pp. 111–116.

[4] Cortesi, D.: An Effective Text-Compression Algorithm. Byte, Vol. 7, 1982, No. 1, pp. 397–403.

[5] CompuServe: Graphics Interchange Format Version 89a. CompuServe Incorporated, Columbus, Ohio, 1990.

[6] Deutsch, P.: DEFLATE Compressed Data Format Specification Version 1.3. Network Working Group, Request for Comments 1951, 1996.

[7] Gage, P.: A New Algorithm For Data Compression. The C Users Journal, Vol. 12, 1994, No. 2, pp. 23–38.

[8] Huffman, D. A.: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the Institute of Radio Engineers, Vol. 40, 1952, pp. 1098–1101.

[9] Jewel, G. C.: Text Compaction for Information Retrieval Systems. IEEE Syst., Man and Cybernetics SOC. Newsletter, Vol. 5, 1976, No. 2, pp. 4–7.

[10] Knuth, D. E.: Dynamic Huffman Coding. Journal of Algorithms, Vol. 6, 1985, pp. 163–180.

[11] Moffat, A.—Neal, R. M.—Witten, I. H.: Arithmetic Coding Revisited. ACM Transactions on Information Systems, Vol. 16, 1995, pp. 256–294.

[12] Nelson, M.—Gail, J.: The Data Compression Book. M & T Books, 1995.

[13] Randers-Pehrson, G.: PNG (Portable Network Graphics) Specification Version 1.2. PNG Development Group, 1999.

[14] Sayood, K.: Introduction to Data Compression. Morgan Kaufmann, San Francisco, 1996.

[15] Schieber, W. D.—Thomas, G. W.: An Algorithm for Compaction of Alphanumeric Data. Journal of Library Automation, Vol. 4, 1971, pp. 198–206.

[16] Snyderman, M.—Hunt, B.: The Myriad Virtues of Text Compaction. Datamation, Vol. 16, 1970, No. 12, pp. 36–40.

[17] Storer, J. A.—Szymanski, T. G.: Data Compression Via Textual Substitution. Journal of the ACM, Vol. 29, 1982, pp. 928–951.

[18] Svanks, M. I.: Optimizing The Storage of Alphanumeric Data. Canad. Datasystems, 1975, May, pp. 38–40.

[19] Welch, T. A.: A Technique for High-Performance Data Compression. IEEE Computer, Vol. 17, 1984, No. 6, pp. 8–19.

[20] Witten, H.—Neal, R. M.—Cleary, R. J.: Arithmetic Coding for Data Compression. Communications of the ACM, Vol. 30, 1987, pp. 520–540.

[21] Ziv, J.—Lempel, A.: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, Vol. 23, 1977, pp. 337–343.

[22] Ziv, J.—Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory, Vol. 24, 1978, pp. 530–536.

Altan Mesut received his B. Sc. degree from Istanbul University, Istanbul, Turkey, in 1998 and his M. Sc. and Ph. D. degrees from Trakya University, Edirne, Turkey, in 2002 and 2006, respectively, all in computer science. From 1998 to 2005 he was a research assistant and from 2005 to 2007 a teaching assistant in Computer Engineering Department of Trakya University. He has been working as an Assistant Professor in this department since 2007. His areas of research include data structure, data compression, data mining and computer graphics.

Aydin Carus received his B. Sc. degree in mathematics in 1991 and his M. Sc. and Ph. D. degrees in computer science in 1994 and 1997, respectively, all from Trakya University, Edirne, Turkey. From 1991 to 1996 he worked as a system analyst in Computer Center of Trakya University and from 1996 to 1997 as a teaching assistant in Computer Engineering Department of Trakya University. He has been an Assistant Professor in this department since 1997. His research interests include data structure, data compression, pattern matching, data mining and software engineering.