A Software Solution for Data Compression Using the Prefix Encoding

Student authors: Ilija J. Urošević[1] and Dejan Ž. Jevtić[1]
Mentors: Radomir S. Stanković[2] and Dušan B. Gajić[2]

Abstract – **Prefix encoding is one of the basic coding systems that use the prefix property. Various applications of the prefix encoding are in use today, e.g. country calling codes, UTF-8 system for encoding Unicode characters, the Secondary Synchronization codes used in the UMTS W-CDMA 3G Wireless standard. In this paper, we first present theoretical bases of prefix encoding and some of the elementary types of prefix coding techniques. Afterwards, we discuss the software implementation of these coding algorithms realized in C# programming language. Experimental results considering data compression ratio and time needed for the compression are also included. The application is primarily developed with educational purposes in mind.**

Keywords – **Data compression, C# application, prefix encoding.**

I. Introduction

Data compression methods are very important in computer science for many reasons, including faster time for data transmission that they allow, and more free space on storage disks. There are many compression methods developed for this purposes.

This paper first describes prefix encoding as one of the basic coding systems and presents the theoretical bases of the prefix coding and some types of prefix codes, described in Sections 2 and 3, respectively. Next, in Section 4 we present a software solution for data compression using the prefix encoding, realized in C# programming language, along with the experimental results representing data compression ratio and time of compression. This application is mainly developed for educational purposes.

II. Prefix Codes

In this Section, we use references [1] and [4]. Prefix encoding is one of the basic and most common coding systems. Prefix code is a variable-type code which satisfies the prefix property. The prefix property requires that once a certain bit pattern has been assigned as the code of a symbol, no other valid code words should start with that pattern (the pattern cannot be prefix of any other code word). For instance, we will define two codes: code-1 with following code words: {1, 01, 010, 001}, and code-2 with code words: {1, 01, 000, 001}. In this case, code-1 does not satisfy the prefix property because code word 010 begins with 01, which is also valid code word in code-1. On the other hand, code-2 does not have an issue with this property, and therefore it satisfies the prefix property. Coding systems with the prefix property can be transmitted with a sequence of code words without any symbol for dividing individual words. If a code does not satisfy the prefix property, situation mentioned above can lead to ambiguous codes. We will demonstrate this with one simple example. Let us define symbols a1, a2, a3 and a4 which we will assign to code-1 and code-2, as shown in Table I:

TABLE I
EXAMPLE OF CODES WITH PREFIX PROPERTY

Symbol	Code-1	Code-2
a1	1	1
a2	01	01
a3	010	000
a4	001	001

We will now code the following sequence of symbols: a1, a3, a2, a1, a4. Code-1 will decode this sequence as 1|010|01|1|001 (without vertical bars). However, this code can be decoded wrongly because of the lack of the prefix property. The decoder doesn't know whether to decode the sequence as 1|010|01… (which is a1, a3, a2,…) or as 1|01|001… (which is a1, a2, a4,…). From this example we can conclude that code-1 is ambiguous. Code-2 has the prefix property and it can be decoded unambiguously.

There are many algorithms which are based on the prefix encoding. Two most well known are the Shannon-Fano method and the Huffman method. Prefix encoding is good solution in situations when we want to code integers because the binary representation of integers does not satisfy the prefix property and size of the set of integers must be known in advance. With prefix encoding we don't have to know size of the set of integers in advance which is required in some applications. There are lot of variations of the prefix codes, and we will describe some of them in the next Section.

Student authors:
[1]Ilija J. Urošević and Dejan Ž. Jevtić are with the Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: kieknai@elfak.rs and dejanjevtic@elfak.rs.

Mentors:
[1]Radomir S. Stanković and Dušan B. Gajić are with the Computational Intelligence and Information Technology Laboratory, Department of Computer Science, Faculty of Electronic Engineering, Aleksandra Medvedeva 14, 18000 Niš, Serbia, E-mails: radomir.stankovic@elfak.ni.ac.rs and dusan.gajic@elfak.ni.ac.rs.

III. Types of Prefix Codes

We based this section mostly on reference [1]. First, we will present the Unary Code. The Unary Code is operating with non-negative integers, and the code of the integer *n* is defined as *n* - 1 ones, followed by one zero. There is also an alternative version of this code. With the alternative version, integer *n* is defined as *n* - 1 zeros, followed by single one. Unary codes are very easy to work with, but disadvantage of these codes is very large size for large numbers, which is not the good solution for compression of integers. Some Unary Codes are represented in Table II.

TABLE II
UNARY CODES

n	Code	Alt. Code
1	0	1
2	10	01
3	110	001
4	1110	0001
5	11110	00001

It is also possible to define general unary codes, also known as start-step-stop codes. This code consists of a triplet (start, step, stop) of integer parameters, and code words are created with the following procedure: *n*th code word consists of *n* ones, followed by one zero, followed by all the combinations of *a* bits where *a* = start + n*step. If *a* = stop, then the single zero preceding *a* bits is left out. The number of codes for the given triplet is finite and depends on the selection of parameters. The example for triplet (3, 1, 7) is shown in Table III. This triplet generates 248 code words.

TABLE III
GENERAL UNARY CODE FOR TRIPLET (3, 1, 7)

n	*a*=3+n*1	*n*th code word	number of code words	range of integers
0	3	0xxx	$2^3 = 8$	0-7
1	4	10xxxx	$2^4 = 16$	8-23
2	5	110xxxxx	$2^5 = 32$	24-55
3	6	1110xxxxxx	$2^6 = 64$	56-119
4	7	1111xxxxxxx	$2^7 = 128$	120-247

As we can see from the table, the number of code words depends on the value of the *a* parameter. With smaller value of *a*, we can present smaller range of integers. As the value of *a* changes, for each group of code words we have a certain prefix. Of course, the prefix property is satisfied.

The number of different general unary codes is given in the Eq. (1).

$$\frac{2^{\text{stop+step}} - 2^{\text{start}}}{2^{\text{step}} - 1}. \qquad (1)$$

This expression increases exponentially with parameter "stop", so large sets of these codes can be generated with small values of (start, stop, step) parameters. For example:

1) The triplet (*n*, 1, *n*) defines the standard *n*-bit binary codes, whose number is given in the Eq. (2).

$$\frac{2^{n+1} - 2^n}{2^1 - 1} = 2^n \qquad (2)$$

2) The triplet (0, 0, ∞) defines the codes 0, 10, 110, 1110,... which are the unary codes but assigned to integers 0, 1, 2, ... instead of 1, 2, 3, ...

3) The triplet (1, 1, 30) produces $(2^{30} - 2^1) / (2^1 - 1)$ codes, which is approximately a billion codes.

Besides the unary codes, there are often prefix codes which are built in a different way. We will explain four of them. Symbol $B(n)$ is used to denote the binary representation of integer *n*, $|B(n)|$ is the length, in bits, of this implementation, and $\overline{B}(n)$ is used to denote $B(n)$ without its most significant bit, which is always 1.

Code C_1 is made of two parts. To code the positive integer *n*, we first generate the unary code of $|B(n)|$ which is the size of the binary representation of *n*, then we append $\overline{B}(n)$ to it. Let us demonstrate this in one example, for $n = 19 = 10011_2$. The size of $B(19)$ is 5, so we start with the unary code 11110 and append $\overline{B}(n)$=0011. The complete code is 11110|0011. Length of code $C_1(n)$ is $2\lfloor \log_2 n \rfloor + 1$ bits.

Code C_2 is a rearrangement of code C_1 where each of the $1+\lfloor \log_2 n \rfloor$ bits of the first part (the unary code) of C_1 is followed by one of the bits of the second part (the $\overline{B}(n)$ part). For $n = 19 = 10011_2$, $C_1(19)$=111100011, after every of first 5 bits we insert one bit from the second part. As a result we have $C_2(19)$=101011110.

Code C_3 is constructed with the following procedure: We start with $|B(n)|$ coded in C_2, and then append $\overline{B}(n)$. For n = $19 = 10011_2$, $|B(19)|$ is 5 bits, then we code this value in C_2, $C_2(5)$=11101 and append $\overline{B}(19)$=0011, and finally we have $C_3(19)$=111010011. The size of $C_3(n)$ is $1+\lfloor \log_2 n \rfloor + 2\lfloor \log_2 (1+\log_2 n) \rfloor$ bits.

Code C_4 consists of several parts. We start with $B(n)$. To the left of this value we write the binary representation of $|B(n)|-1$ (the length of *n*, minus 1). This procedure continues recursively, until a 2-bit number is written. A zero is then added to the right of the entire number to make it decodable. For n = $19 = 10011_2$, we start $B(19)$=10011, then add the binary representation of $|B(19)-1|=4$ which is 100_2, and we have 100|10011. In the next step, we add the binary representation of $|B(4)-1|=2$ which is 10_2, add it to right of the current result, and we get 10|100|10011. Finally, we add a zero to the right of current result, and as the final result we have $C_4(19)$=10|100|10011|0.

The length of these four codes increases as $\log_2 n$ while the length of unary code increases as *n*. These codes are therefore good choices in cases where the data consists of integers *n* with probabilities that satisfy certain conditions.

IV. Software Implementation and Experimental Results

For software implementation, we used references [2], [3] and [5]. The software solution for prefix encoding that we implemented is "Prefix Coder". This application can encode data using unary code and codes C_1 and C_2. "Prefix Coder" is developed in C# programming language and .NET Framework 3.5. The architecture of the application is shown in Fig. 1.

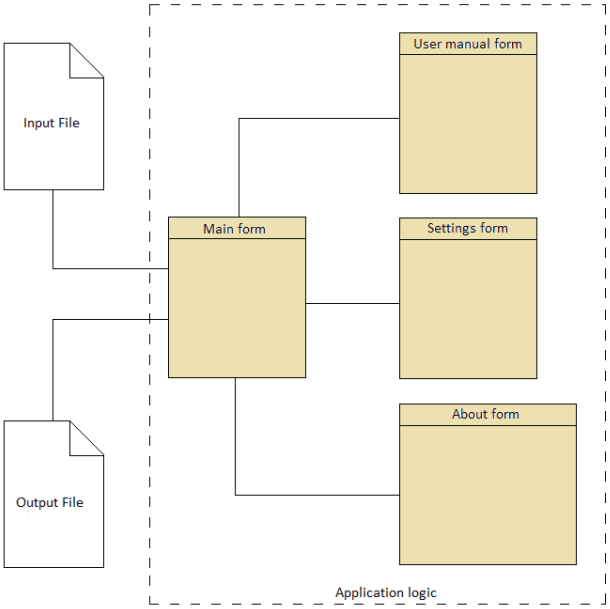

Fig. 1. "Prefix Coder" architecture.

Prefix coder consists of Main form, Settings form, About form and User manual form. Main form represents the starting window of the application and it's linked with 3 other forms. Settings form is used for simple settings of the application, like separators that are used or default input name etc. User manual form contains instructions for using the "Prefix Coder", and the About form has a description of the application and authors. These four forms together make the application logic. Main form is also connected with input and output files.

Input data of the application can be inserted via text field or a text file. Possible outputs are a text field, a text file or a binary file. Text field and text file as outputs are used for representation of the coding results. If the selected output method is binary file, after the encoding process is finished the compression rate is shown.

When the encoding process starts, first the input data is read from the selected input and stored in array of strings, where each string is a number that needs to be encoded. Next, that data is forwarded to one of the functions which implements a coding algorithm for the selected coding method. All of these functions generate an array of BitArrays. In this array is the sequence of ones and zeros that is the result of the coding. All of the functions call SaveCode function which saves the data in the way that is selected as output method. Since the C# language doesn't have an interface that allows for single bits to be written into a file, all the coded data is first transferred into bytes and written into files in that form. This also means that neutral characters (zeros) will be added to last chunks of data until the chunk is 8-bit long.

Interface of the "Prefix Coder" is designed to be intuitive and easy to use and it is shown in Fig. 2.

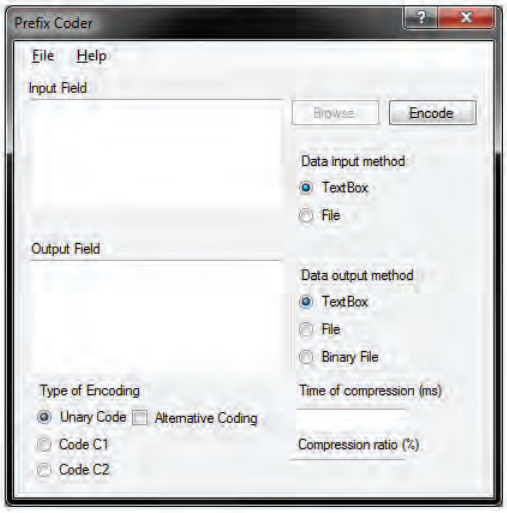

Fig. 2. Interface of the "Prefix Coder".

The Settings form of the "Prefix Coder" is shown in Fig. 3. In this form we can adjust properties of the "Prefix Coder". These properties are Tooltip time (display time of tooltip), Separators, Default path for the output file and Default name for the output file. These properties are initially set to optimal values. These properties are kept in a separate file. Separators are used to separate numbers in the Input Field when "Textbox" is selected as Data input method. User can define his own separator. In order to do this, he must follow certain rules. The default separator is white space. If user wants to add a separator, after entering desired symbol he must add character "s" without making any spaces. If user does not want to use white space as a separator, he must first clear Separators field and then add his separators in a procedure described above.

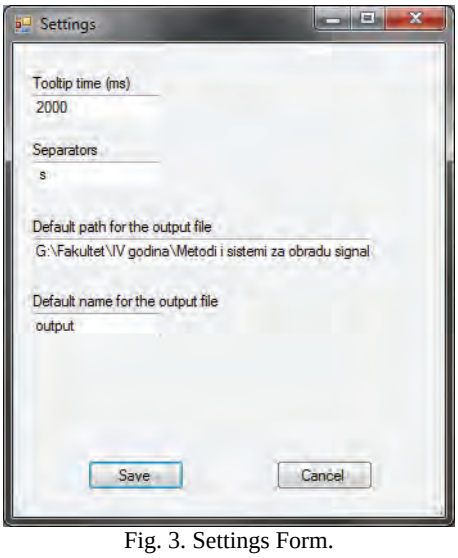

Fig. 3. Settings Form.

The experimental results considering time needed for compression are shown in Table IV. For all the data in Table IV tests were performed several times and calculated average values of the results are taken as final results. It should also be noted that the duration of the compression depends on the computer hardware and current utilization of computer resources. These tests were performed on the AMD Phenom(tm) 8450 with 2 GB of DDR2 RAM and on Windows 7 64-bit OS.

TABLE IV
TESTS FOR TIME OF COMPRESSION

	Time of compression (ms)		
Input data	Unary code	Code C_1	Code C_2
0-100	~1	~1	~1
500	~5	~1	~1
10000	~400	~1	~1
10 two-digit integers	9	3	3
10 four-digit integers	22500	8	8
10 five-digit integers	350000	10	10

From the Table IV it can be seen that the compression with the unary coding needs much more time than with the codes C_1 and C_2, since the length of the unary code increases linearly, and the compression time is longer for one larger integer number than for several smaller ones.

In Table V experimental results for compression ratio are shown. As mentioned before, C# language doesn't have the interface for writing single bits in a file, all the output data have several bits more than it would have if this restriction doesn't exist.

TABLE V
TESTS FOR COMPRESSION RATIO

		Size of the output file (Byte) / compression ratio		
Size of input file (Byte)	Input Data	Unary code	Code C_1	Code C_2
5	500	63 /0%	3 /40%	3 /40 %
5	5,5,5	2 /60%	2 /60%	2 /60%
1	5	1 /0%	1 /0%	1 /0%
20	5,5,5,5,5,5,5,5,5,5	6 /70%	6 /70%	6 /70%
11	500,500,500	187 /0%	7 /36%	7 /36%

Table V shows that compression ratio increases as the number of the parameters increase, and decreases with larger value of numbers in input data.

V. CONCLUSION

In real-world circumstances, the unary code does not give satisfying results, but it is very important because of the idea of prefix encoding. For specific values of input data, it takes a very long time for program execution, and no compression is achieved.

Codes C_1 and C_2 give results which are mostly satisfying and similar. For specific values of input data a very high compression ratio can be achieved. The time of compression is very short in most of the cases. Our application implements the unary code, and both codes C_1 and C_2, and it is developed mainly for educational purposes.

REFERENCES

[1] D. Salomon, *Data Compression – The Complete Reference*, Springer, 2007.

[2] *Form Class*, http://msdn.microsoft.com/en-us/library/system.windows.forms.form%28VS.71%29.aspx, website last visited on 4/4/2011.

[3] *Measuring Execution Time in C#*, http://www.codersource.net/microsoft-net/c-miscellaneous/measuring-execution-time-in-c.aspx, website last visited on 4/4/2011.

[4] *Prefix code - Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Prefix_code, website last visited on 4/4/2011.

[5] *BitArray Class (System.Collections)*, http://msdn.microsoft.com/en-us/library/system.collections.bitarray.aspx, website last visited on 4/4/2011.