Computer Standards & Interfaces 37 (2015) 53–59

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Rapid lossless compression of short text messages

Kenan Kalajdzic [a,*], Samaher Hussein Ali [c], Ahmed Patel [a,b]

[a] School of Computer Science, Centre of Software Technology and Management (SOFTAM), Faculty of Information Science and Technology (FTSM), Universiti Kebangsaan Malaysia, UKM Bangi, 43600 Selangor Darul Ehsan, Malaysia
[b] School of Computing and Information Systems, Faculty of Science, Engineering and Computing, Kingston University, Penrhyn Road, Kingston upon Thames KT1 2EE, United Kingdom
[c] Department of Information Network, Faculty of Information Technology (IT), University of Babylon, Babylon 00964, Iraq

ARTICLE INFO

Article history:
Received 28 November 2012
Received in revised form 27 May 2014
Accepted 28 May 2014
Available online 6 June 2014

Keywords:
Data compression
Lossless compression
Short text messages
SMS

ABSTRACT

In this paper we present a new algorithm called b64pack[1] for compression of very short text messages. The algorithm executes in two phases: in the first phase, it converts the input text consisting of letters, numbers, spaces and punctuation marks commonly used in English writings to a format which can be compressed in the second phase. The second phase consists of a transformation which reduces the size of the message by a fixed fraction of its original size. We experimentally measured both the compression speed and the compression ratio of b64pack on a large number of short messages and compared them with compress, gzip and bzip2, three most common UNIX compression programs. We show that in case of short text messages up to a certain size b64pack achieves better compression than any of the three programs. With respect to speed, b64pack beats all three algorithms by orders of magnitudes. This rapid compression is one of the key strengths of b64pack.

1. Introduction

Until recent years, most algorithms for text compression were primarily concerned with compressing large inputs. Fast adoption of SMS messaging and Internet services based on short messages (e.g. Twitter, chat) has caused an increased interest in compression of very short texts. Interestingly, though, publications concerning compression of short messages are relatively scarce.

Why is compression of short messages necessary? Given the high volume of SMS, Twitter and instant messaging traffic, compression of short text messages can bring tremendous savings in network bandwidth. Could not multiple messages be first buffered to form a larger chunk of data and then compressed with a regular compression algorithm to achieve better results? The answer is: For realtime communication, such as instant messaging or chat, buffering of multiple messages is not possible, since each message has to be sent independently and immediately after it is typed. Therefore we need a mechanism to compress each of these short messages individually.

In case of SMS messages, a system called *concatenated SMS* has been developed to extend the inherent limit of an SMS message. It works by breaking a long message into smaller parts and sending each of them as a single SMS message. At the receiving end the short messages are combined back to one long message. One downside of concatenated SMS is that, if the length of an SMS message exceeds 140 bytes, the user is usually charged for two SMS messages, even if the excess is only a few characters long.

In this paper we introduce a new algorithm called b64pack for efficient compression of very short text messages. In contrast with other major works in short text compression, such as [1–3], which focus on certain limitations of *prediction by partial matching* (PPM) compression and provide ways to improve it, we follow a different approach.

To facilitate an easy deployment and interoperability across billions of computers, mobile and embedded devices, we propose a compression scheme which relies on a straightforward use of standard open source software libraries available on all operating systems. The use of b64pack does not require any proprietary software components or algorithms. We compare b64pack with other standard compression algorithms implemented by programs such as compress, gzip and bzip2 to demonstrate how applications and users could directly benefit from using b64pack for compression of short messages. Our research objective was to prove that b64pack is able to overcome certain major drawbacks of existing SMS services. We did not specifically set out or purport to evaluate against other data compression schemes, and have used them merely as a reference for comparison.

The key features of b64pack are:

- extremely low memory requirements—a message compressed with b64pack requires no header/metadata, while in the base case lookup tables used by b64pack together occupy less than 256 bytes of memory;

* Corresponding author.
E-mail addresses: kenan@unix.ba (K. Kalajdzic), samaher@itnet.uobabylon.edu.iq (S.H. Ali), whinchat2010@gmail.com (A. Patel).

[1] b64 stands for BASE64.

[1] b64 stands for BASE64.

http://dx.doi.org/10.1016/j.csi.2014.05.005
0920-5489/

- very efficient compression and decompression—all operations performed by `b64pack` can be implemented very efficiently using few CPU instructions, allowing `b64pack` to be used for realtime message compression on low-power devices with energy consumption limitations;
- precise estimation of the size of the compressed message during the compression process—this feature allows users to know the size of the compressed message while they are composing it;
- reliance on standard software libraries to facilitate rapid deployment and interoperability on all types of computers, mobile and embedded devices.

Other than the benefits mentioned above, b64pack is a fast process-oriented algorithmic scheme which we believe should be considered by developers, users and standards setting bodies as a viable compression technique.

The plan for the remainder of the paper is as follows: in Section 2 we describe `b64pack` algorithm in detail. In Section 3, we provide experimental data of performance and compare it with those of three well known algorithms implemented by `compress`, `gzip` and `bzip2`. Sections 4 and 5 deal with discussion, future work and conclusions.

2. The `b64pack` algorithm

As illustrated in Fig. 1, the `b64pack` algorithm consists of two phases. The primary purpose of the first phase is to convert the input text to a format which can be processed in the second phase. The input can optionally be precompressed in this first phase to achieve higher gross space savings. We assume that the input is a short text message, consisting of letters, numbers, spaces and punctuation marks commonly used in English writings. Even though there are no inherent limitations imposed on the nature of the input, we demonstrate the workings of `b64pack` by following the compression of an SMS message. Therefore, we assume that the input text contains only those punctuation marks, which have a definition within the GSM 03.38 character set [4].

The output generated by the first phase is processed in the second phase, which consists of a single transformation that reduces the size of the message by a fixed percentage. This step is thus fully deterministic and always results in the same, constant compression ratio.

An important characteristic of the whole `b64pack` compression procedure is the absence of any metadata. This means that the compressed message requires no header, which is highly important when working with SMS messages or similar kinds of short texts which are inherently limited to a small number of characters.

2.1. Message transcoding

The compression, which happens in the second phase of `b64pack` algorithm, requires the input text to be transformed to a specific format. To achieve this, the input is transcoded using the following simple rules:

Rule 1 *Letters and numbers are left unchanged.*

Rule 2 *Each* SPACE *character is replaced with a forward slash '/' character.*

Rule 3 *Each punctuation mark is replaced with a sequence of two characters: the plus '+' character followed by a lowercase letter. The correspondence between punctuation marks and their substitute lowercase letters is established through* Table 1.

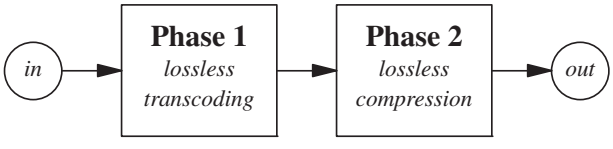

Fig. 1. The phases of the `b64pack` compression algorithm.

Table 1
Mapping of punctuation marks to letters.

Character	@	$	_	!	"	#	%	&	'	(	)	*	+
Substitute	a	b	c	d	e	f	g	h	i	j	k	l	m
Character	,	-	.	/	:	;	<	=	>	?	□	□	□
Substitute	n	o	p	q	r	s	t	u	v	w	x	y	z

□ = reserved for future use.

Rule 3 applies to most common punctuation marks, for which there is a single-character code in the GSM 03.38 character set. For less frequently used punctuation marks, GSM 03.38 provides another representation consisting of two characters per punctuation mark (first of these two characters is the escape character `0x1b`). For these we use the following rule in place of Rule 3:

Rule 4 *Each punctuation mark from the set of characters shown in the first row of* Table 2 *is replaced with a sequence of three characters: two plus '+' characters followed by a corresponding letter from the second row of* Table 2.

To show how the transcoding procedure alters the input, we use the example message given in Fig. 2. The length of this message is exactly 160 characters, which is the limit imposed on an SMS message with 7-bit encoding. For clarity, spaces are represented by white boxes.

Following the aforementioned rules for transcoding, we transform this message into the form shown in Fig. 3.

The use of the '+' escape character for encoding punctuation marks has led to an increase in message length from 160 to 173 characters. To reduce this loss, we make use of a simple typographic rule, which states that it is often appropriate to insert a space after punctuation in order to increase the overall readability of the text. This typographic convention has been used multiple times in our example message (Fig. 2). Based on this observation we can now introduce another simple encoding rule:

Rule 5. *If a punctuation mark is followed by a* SPACE*, this punctuation mark is encoded according to Rule 3 or Rule 4, except that instead of a lowercase letter its uppercase equivalent is used. In this case, the encoded* SPACE *character (i.e., the forward slash character) is left out.*

For example, according to Rule 3, a question mark followed by a SPACE would be encoded as '`+w/`'. Rule 5 allows both these characters to be merged into a two-byte sequence '`+W`', thus preventing any loss caused by the use of the '+' escape character.

Following the same procedure, the whole message can be transformed into the form shown in Fig. 4. The length of this message is 167 characters.

2.2. Compression of the transcoded message

A closer look at the transcoded message in Fig. 4 reveals an important clue. Namely, all the characters which appear in this message are part of the BASE64 character set.

BASE64 encoding maps an arbitrary sequence of 24 bits into a sequence of four printable characters. In a given sequence of 24 bits (i.e., three octets) of data

$$a_1a_2a_3a_4a_5a_6a_7a_8 \;\; b_1b_2b_3b_4b_5b_6b_7b_8 \;\; c_1c_2c_3c_4c_5c_6c_7c_8$$

Table 2
Mapping of less frequent punctuation marks to letters.

Character	[	\	]	^	{	|	}	~
Substitute	a	b	c	d	e	f	g	h