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.

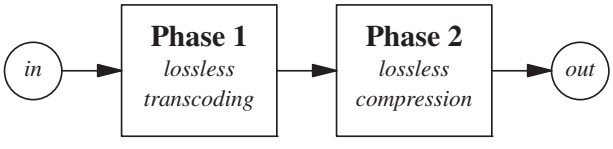

Fig. 1. The phases of the `b64pack` compression algorithm.

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

```
Hi□mom!□Just□wanted□to□let□you□know□that□John□moved□to
□New□York□last□Tuesday.□How□are□you?□By□the□way,□I've □
got□a□new□phone□number:□+387(87)654321.□Love□you□:-)
```

Fig. 2. An example message to be compressed using `b64pack`.

the individual bits are grouped to form four bit sextets

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

Each sextet is then replaced with a printable character based on the mapping established through Table 3.

BASE64 decoding is the reverse operation, which uses Table 3 to convert a block of four characters into an equivalent block of three octets.

BASE64 encoding was first introduced in Internet Engineering Task Force (IETF) *de facto* standard RFC989 [5] to allow binary data to be represented in printable form. In the usual case, BASE64 is used to convert three octets of data into four 8-bit Extended ASCII characters. This means that each of the characters in even columns of Table 3 is represented with 8 bits.

There is, however, no specific requirement that these characters be coded using 8 bits. We can make use of this fact to compress the text of an SMS message using BASE64 decoding, by treating the text in Fig. 4 as BASE64-encoded data.

2.2.1. Compression ratio for input text with 7-bit characters

If we are dealing with an SMS message encoded using the GSM 03.38 character set, each character in Fig. 4 is represented using 7 bits. A group of four characters thus occupies 28 bits.

Applying BASE64 decoding to a four-character block reduces its size to 24 bits. The compression ratio is defined as the fraction where the numerator is the size of the compressed message and the denominator is the size of the original message with 7 bits/character. Thus we have

$$ratio_{7bit} = \frac{24}{28} = \frac{6}{7} \approx 85.7\% \tag{1}$$

Given the fact that the maximum size of an SMS message is 160 characters, based on (1) we can now calculate the maximum length L_{tmax} of a transcoded message (i.e., SMS message after processing in phase 1 of `b64pack`) which would not exceed 160 characters after BASE64 decoding:

$$L_{tmax} = 160 \times \frac{7}{6} \approx 186.67 \approx 186 \text{ characters} \tag{2}$$

The maximum length of 186 characters, calculated in (2), applies to an already transcoded SMS message. As we know from Section 2.1, transcoding of punctuation marks may increase the size of the original message, so that the user would have to estimate in advance how much to type in order to stay within the limits of a single SMS message after compression.

Although it is not possible to know in advance how much space in a message would be occupied by punctuation, a simple statistical analysis of a standard corpus of English writings [6] shows that punctuation takes around 2.67% of the corpus text. We could thus assume that nearly 3% of the length of user input are punctuation marks.

Assuming that every punctuation mark extends the original message by one character, transcoding would extend the user message by 3%. If we denote the length of the original user message by L_u, its length after transcoding would be

$$L_t = L_u \times 1.03 \tag{3}$$

To calculate the maximum length L_{umax} of the user message, which after compression with `b64pack` should fit into a single SMS message, we substitute (2) into (3):

$$L_{umax} = \frac{L_{tmax}}{1.03} = \frac{186}{1.03} \approx 180.58 \approx 180 \text{ characters} \tag{4}$$

This means that in most cases the sender of an SMS message should be able to type 180 characters and stay within the 140-octet limit of a single SMS message after `b64pack` compression has been performed.

2.2.2. Compression ratio for input text with 8-bit characters

The calculation of compression ratio given in (1) is valid for SMS messages using 7-bit character encoding. In case `b64pack` compression is applied to text messages encoded in 8-bit Extended ASCII (e.g. instant messages, IRC messages, Twitter tweets, RSS feeds), the corresponding ratio is

$$ratio_{8bit} = \frac{24}{32} = \frac{3}{4} = 75\% \tag{5}$$

2.3. Runtime estimation of maximum message size

Using BASE64 decoding in the second phase of `b64pack` allows the compression to happen while the user is typing the message. An application for message composition can begin transcoding the message as soon as the user begins typing. Each time the transcoding function generates four characters, these four characters can immediately be processed by the BASE64 decoder to produce the corresponding sequence of three octets of compressed data.

For example, if the user is composing the message shown in Fig. 2, as soon as the user types '`Hi m`', `b64pack` transcodes this to '`Hi/m`' and then proceeds with BASE64 decoding which converts these four characters to the three-octet sequence `0x1e`, `0x2f`, `0xe6`.

In case of SMS, this allows an SMS composition application to calculate the number of bytes occupied by the compressed message and provide the user with an estimated number of characters left for input before the 140-octet limit for a single SMS message is reached.

2.4. Dictionary substitution

The two phases of `b64pack` compression, described in Sections 2.1 and 2.2, guarantee a fixed, deterministic compression ratio. Introducing

```
Hi/mom+d/Just/wanted/to/let/you/know/that/John/moved/to
/New/York/last/Tuesday+p/How/are/you+w/By/the/way+n/I+ive/
got/a/new/phone/number+r/+m387+j87+k654321+p/Love/you/+r+o+k
```

Fig. 3. Example message from Fig. 2 after applying Rules 1, 2 and 3.

```
Hi/mom+DJust/wanted/to/let/you/know/that/John/moved /to
/New/York/last/Tuesday+PHow/are/you+WBy/the/way+NI+ ive/
got/a/new/phone/number+R+m387+j87+k654321+PLove/you /+r+o+k
```

Fig. 4. Example message from Fig. 2 after transcoding.

a simple dictionary, shared between the sender and the receiver, can lead to further space savings.

To save additional space, every word in the input text, for which there is an entry in the dictionary, is replaced with the corresponding key. More generally, dictionary substitution can be applied to an arbitrary sequence of letters.

For `b64pack` dictionary substitution, a key is formed by concatenating the plus character '+' and a sequence of numbers which uniquely corresponds to the replaced sequence of letters. If a letter sequence is found in the dictionary, it is replaced with the corresponding key. This happens immediately after the transcoding phase, before BASE64-decoding takes place (Fig. 5). Obviously, to achieve space savings, the total length of the key must be at least one character less than the length of the replaced sequence.

It is important to notice that this method of forming the key does not allow substitution in case of letter sequences immediately followed by numbers. If, for instance, the key for the word '`high`' is '`+71`', then the sequences '`highfive`', '`high/five`' and '`high/5`' can safely be replaced with '`+71five`', '`+71/five`' and '`+71/5`' respectively. On the other hand, using the same key in case of the sequence '`high5`' would produce '`+715`', which is ambiguous.

2.4.1. An example of dictionary substitution

To see dictionary substitution in action, let us assume the words (letter sequences) *are*, *know*, *last*, *one*, *that*, *the*, *want* and *you* exist in the dictionary. The dictionary maps each of these letter sequences to a unique sequence of numbers, as shown in Table 4.

Now we can revisit the transcoding phase and apply dictionary substitution to the transcoded message in Fig. 4. Fig. 6 shows the result of this transformation.

In this example, dictionary substitution reduced the length of the transcoded message from 167 to 157 characters. This fully compensates for the loss introduced by transcoding and saves three additional characters (recall that the original message size was 160 characters), which is a significant gain given the tiny size of the dictionary.

2.4.2. Building and sharing the dictionary

A dictionary is built from a list of most frequent words appearing in a large set of messages. The building of a dictionary can be a one-time process, or can be done periodically to adapt to possible changes in word frequencies. Which of these two dictionary-building approaches is used, largely depends on the type of application for which it is used.

If `b64pack` compression is to be used on a mobile phone for exchange of SMS messages, the simplest approach consists of building the dictionary once and bundling it with the SMS composition application. This way the dictionary would be installed along with this application, which guarantees that all its users share the same dictionary. In a massive deployment of such an application, it is impractical to allow dictionary to be updated, since this may introduce various inconsistencies when multiple users have different versions of the dictionary.

More flexibility with regard to updating dictionary contents can be achieved in cases where the dictionary is shared between a small number of devices, especially if these devices operate under common administration. In private communication, it may be feasible to have a dictionary shared between only two parties, if these two parties exchange a relatively high number of messages. In a company, the dictionary may be updated as a part of a policy which affects all users.

The situation is even more convenient with messaging applications which work over IP networks. These applications can easily have multiple versions of the dictionary and can perform regular dictionary updates. In case of server-based instant messaging or chat applications, such as Jabber or IRC, the server could easily push new versions of the dictionary to all logged-in clients.

Table 3
BASE64 code table.

Pattern	Char	Pattern	Char	Pattern	Char	Pattern	Char
000000	A	010000	Q	100000	g	110000	w
000001	B	010001	R	100001	h	110001	x
000010	C	010010	S	100010	i	110010	y
000011	D	010011	T	100011	j	110011	z
000100	E	010100	U	100100	k	110100	0
000101	F	010101	V	100101	l	110101	1
000110	G	010110	W	100110	m	110110	2
000111	H	010111	X	100111	n	110111	3
001000	I	011000	Y	101000	o	111000	4
001001	J	011001	Z	101001	p	111001	5
001010	K	011010	a	101010	q	111010	6
001011	L	011011	b	101011	r	111011	7
001100	M	011100	c	101100	s	111100	8
001101	N	011101	d	101101	t	111101	9
001110	O	011110	e	101110	u	111110	+
001111	P	011111	f	101111	v	111111	/

3. Results and evaluation

To measure the two most important aspects of compression of short text messages with `b64pack`, namely, compression speed and compression ratio, we ran multiple experiments under the GNU/Linux operating system running on a general-purpose PC with 1 GB of RAM and an Intel Pentium 4 HT CPU running at 3.0 GHz.

In all the experiments, `b64pack` was compared with `compress`, `gzip` and `bzip2`, three most common UNIX compression programs. `Compress` is based on a variant of the LZW compression algorithm, `gzip` uses the Deflate algorithm, while `bzip2` relies on Burrows–Wheeler compression and Huffman coding.

We first considered running our experiments on parts of the Canterbury Corpus [7], which is the most commonly used reference corpus for evaluating and comparing compression algorithms. Since, however, we are dealing with compression of short text messages, whose structure and typography greatly differ from those of literary works and long texts, such as those available in the Canterbury Corpus, we had to adopt another set of texts, which resemble true structure of messages we wish to compress.

Fortunately, researchers from the Web Information Retrieval/Natural Language Processing Group of the National University of Singapore have compiled a corpus of a huge number of real SMS messages (NUS SMS Corpus) contributed by different people [8]. In all our experiments, we used 50,619 messages from the corpus (including duplicates), which contain exclusively Latin characters.

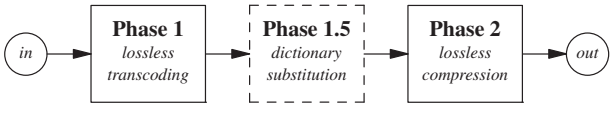

Fig. 5. The phases of `b64pack` with dictionary substitution.

Table 4
Mapping of letter sequences to numbers in the dictionary.

Sequence	Mapped to	Sequence	Mapped to
The	0	want	08
One	1	that	23
Are	3	know	65
You	7	last	84

3.1. Measurements of compression speed

The objective of our first set of experiments was to compare `b64pack` with the other three compression methods with regard to compression speed. Since we were dealing with very small amounts of data (i.e., short messages), it was difficult to make speed comparisons by measuring times required for compressing each individual message. Instead, we let each algorithm compress the entire set of 50,619 messages and measured the total compression time for each of the four algorithms, then divided these times by the total number of messages to obtain the average compression time for a single message.

The results of these measurements are summarized in Fig. 7. They clearly demonstrate superiority of `b64pack`, which is almost twice as fast as `compress`, about 5.5 times faster than `gzip` and about 9 times faster than `bzip2`.

3.2. Measurements of compression ratio

When it comes to compression of short text messages, the obvious advantage of `b64pack` over `compress`, `gzip` and `bzip2` lies in the fact that `b64pack` does not require a header. Otherwise, all the other three compression methods are far superior to what `b64pack` is able to achieve by the simple transformations it performs.

Since we are primarily interested in compression of SMS messages, we have concentrated our experiments around those messages, whose length is greater than 160 characters. There are a total of 1984 such messages in the NUS SMS Corpus. Since the primary goal of compression of these SMS messages is to make them fit into the 160-character limit, in our experiments we compressed each of the messages with each of the four algorithms and counted the successful compressions resulting in the compressed message being less than or equal to 160 characters in length. Even though `b64pack` can only successfully compress messages with lengths up to 213 characters ($160 \times \frac{4}{3} \approx 213.33$), we haven't imposed any limits on the maximum message length in our experiments, whose results are shown in Table 5.

The total number of successful compressions is 1006. It is obvious from Table 5 that `gzip` is the only true competitor to `b64pack` when it comes to compressing short messages. Nonetheless, our results show that `b64pack` is still significantly superior to `gzip` in the total number of successfully compressed messages. In most cases `b64pack` achieves minimal compression ratio and is the only successful algorithm out of all four in about 21%of all successful compressions, with `gzip` being the only successful algorithm in around 4% of all successful compressions.

Since the compression ratio of `b64pack` depends on the structure of the message (e.g. more punctuation means worse compression), it is interesting to know how successful `b64pack` is in compressing real SMS messages of varying lengths. Fig. 8 shows the length distribution of the 947 messages successfully compressed by `b64pack`.

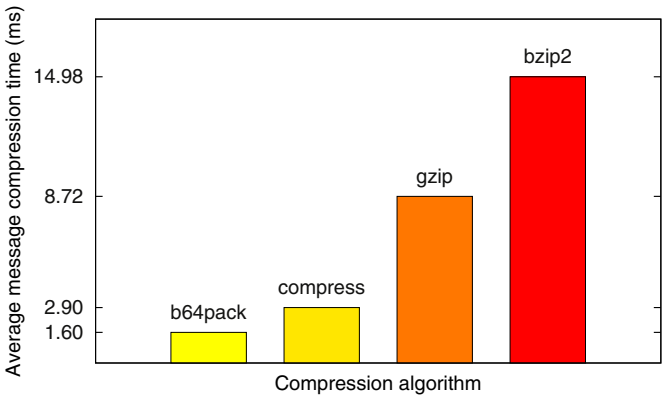

Fig. 7. Compression time comparison between `b64pack`, `compress`, `gzip` and `bzip2`.

As Fig. 8 shows, `b64pack` is good at compressing real SMS messages of all lengths from 161 to 213 characters, being slightly more successful in the case of shorter messages with lengths ranging up to about 180 characters.

A further experiment shows how the use of a simple and relatively small dictionary affects the compression ratio of all four considered algorithms, in particular `b64pack`. For this experiment we used a dictionary containing a total of 1110 frequent English words: 10 three-letter words, 100 four-letter words and 1000 five-letter words. The runtime storage requirements for this small dictionary can be accommodated with 6 kilobytes of memory. As a reference for building the dictionary we used a list of 10,000 most frequent English words obtained from [9].

In case of `compress`, `gzip` and `bzip2`, we first performed dictionary substitution on the original SMS messages, then let each algorithm perform compression and counted successful compressions. In case of `b64pack`, dictionary substitution was performed after transcoding, as described in Section 2.4.

Our results are summarized in Table 6. With dictionary precompression the total number of successful compressions grew from 1006 to 1179. This does not look like a significant gain, yet for the individual algorithms, dictionary precompression has certainly brought some benefits.

Even though `compress` is the winner in the absolute sense in terms of additionally performed successful compressions, `b64pack` and `gzip` still retained more significance. We must, however, notice that `b64pack` gained 20% in additional successful compressions, whereas the relative gain of `gzip` is 10%. Other evaluation criteria in Table 6 show a loss of `gzip` over `b64pack`.

Finally, we wanted to see how dictionary precompression affected the maximum message length and the number of messages of different lengths successfully compressed by `b64pack`. Fig. 9 shows the results. On the graph, we have also included the results we got without dictionary precompression as given in Fig. 8.

As we can see, in our experiments dictionary precompression helped increase the potential maximum message length to around 230, which corresponds to a gain of approximately 10%.

```
Hi/mom+DJust/+08ed/to/let/+7/+65/+23/John/moved/to
/New/York/+84/Tuesday+PHow/+3/+7+WBy/+0/way+NI+ive/
got/a/new/ph+1/number+R+m387+j87+k654321+PLove/+7/+ r+o+k
```

Fig. 6. Example message from Fig. 2 after transcoding and dictionary substitution.

Table 5
Results of experiments comparing success of individual algorithms in compressing messages longer than 160 characters.

Algorithm	b64pack	compress	gzip	bzip2
Total successful compressions	947	467	793	204
Only one algorithm succeeds	212	0	41	0
Minimal compression ratio	886	0	109	0

Table 6
Results of experiments comparing success of individual algorithms in compressing messages longer than 160 characters after dictionary substitution.

Algorithm	b64pack	compress	gzip	bzip2
Total successful compressions	1140	709	874	395
Additional successful compressions	193	242	81	191
Only one algorithm succeeds	299	0	19	0
Minimal compression ratio	1102	4	71	0

4. Discussion—future work

In the average case, message transcoding and BASE64-decoding produce a nearly fixed ratio for all message lengths. Therefore, we propose that future research addresses dictionary precompression. We have seen how a relatively small dictionary with 1110 entries, based on a generic list of words, helped increase maximum length of an SMS message by about 10%. Richer dictionaries with longer words would lead to further improvements in the overall compression ratio of b64pack. With access to a large set of real messages, such as those available in the NUS SMS Corpus, the dictionary could be optimized to include most frequent words from true conversations. This should give maximal space savings in the dictionary substitution phase.

Besides enhancing dictionary substitution, improvements in compression speed can easily be achieved by parallelizing the two phases of b64pack. The fact that the BASE64 decoder operates on groups of four characters allows it to begin decoding as soon as the transcoder generates a sequence of four characters. We saw an example of this in Section 2.3. If transcoding and BASE64-decoding functions are executed in parallel, we can expect to see measurable speed improvements on hardware which allows simultaneous execution of two or more threads.

The b64pack algorithm is very similar to the class of compression techniques known as *transformation algorithms* that perform a reversible precompression transformation which increases its compressibility to other compression techniques. The b64pack algorithm is distinguished from these algorithms by the facts that the transform itself provides a fixed compression ratio and there is no need for metadata to be transmitted along with the compressed file.

The well known Burrows–Wheeler Transform [10,11] is a reversible sorting algorithm that creates long runs of identical characters in the output text. This is followed by a Move To Front algorithm and by Huffman or arithmetic coding. This yields compression that outperforms most of the classic algorithms. A much simpler recent algorithm proposed by Mukherjee and Franceschini called *Star Compression* performs a transformation using a dictionary and yields comparable performance with respect to compression [12]. Its implementation is available on-line [13]. Several modifications have been proposed to improve the performance (see [14]). We plan to explore the possibility of applying the Star Compression ideas for SMS messages.

Finally, one additional strength of b64pack lies in the fact that it relies on standards and *de facto* standards, such as SMS code tables and BASE64 encoding. BASE64 decoding, which forms the basis of the second phase of b64pack compression, is well supported on all operating systems and is available in all major programming languages in form of standard library functions and classes. In the interest of increased interoperability, future research should attempt to find new solutions to advance the process of large-scale deployment of new compression algorithms. To avoid potential legal issues and reduce costs in massive deployments, such as those among mobile phone users worldwide, these advancements should follow the track of standard-compliance and avoid the use of proprietary algorithms.

5. Conclusion

In this work we have shown how a combination of relatively simple transformations can be used to efficiently compress text. More importantly, we proved that for compression of short texts this approach is superior to other well-known lossless compression algorithms. One major advantage of b64pack in the context of compression of short texts lies in the fact that it does not require any metadata. This effectively means that the compressed message can be transmitted without a header, which keeps the full capacity of the communication link for transmission of the data. This is essential for some types of short messages which are inherently limited in size, such as SMS messages. The absence of a header together with the fact that b64pack uses extremely small amounts of memory space and CPU processing power, makes b64pack a very powerful method which can hardly be beaten in compressing short messages by any of the existing compression algorithms.

Compression of short texts can be employed in a wide range of important applications, such as SMS, instant messaging, IRC, Twitter, RSS, and many more. Given the huge volume of such messages exchanged today on the Internet and within GSM networks, bandwidth savings which can result from deployment of b64pack can be tremendous. Furthermore, the ability to compress short messages at almost no computational cost can potentially conserve huge amounts of energy consumed by wireless communication between computers and sensors in Wireless Sensor Networks and large networked embedded systems such as the Internet of Things.

Very efficient and lightweight processing and low memory requirements of b64pack make it particularly suitable for use on mobile phones. B64pack gives mobile users the privilege of typing up to 213 characters for the price of a single SMS message. We saw how this

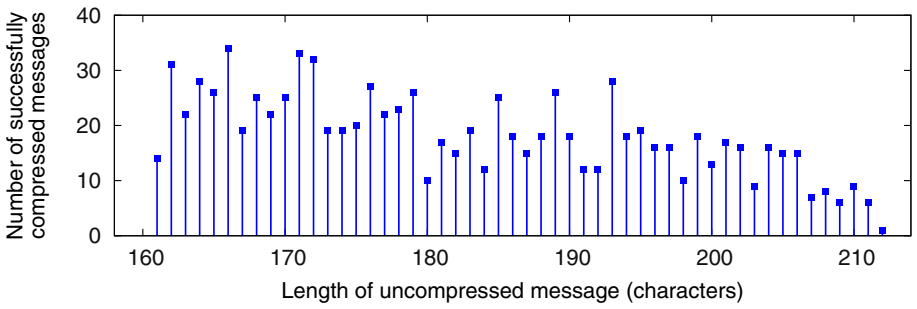

Fig. 8. Distribution of successful compressions by b64pack over different message lengths.

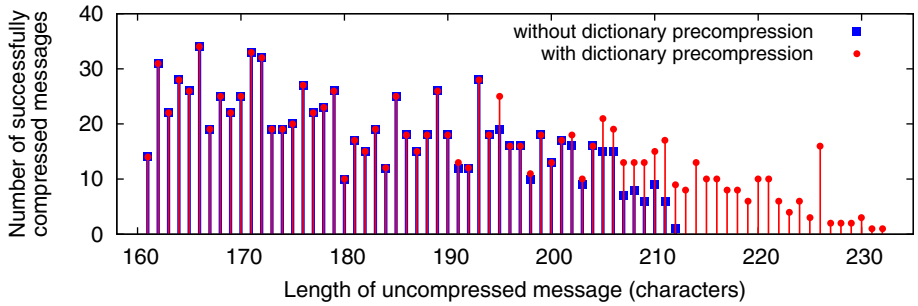

Fig. 9. Comparison of distributions of successful compressions by `b64pack` over different message lengths with and without dictionary precompression.

limit can easily be extended up to about 230 characters by using a small static dictionary with a size of a few kilobytes only.

Another important aspect of `b64pack` compression is that it can be done in real time, while the user is typing a message. We have described how this fact can be used to estimate the length of the compressed message during composition.

6. Acknowledgments

The authors wish to thank Professor Amar Mukherjee from the University of Central Florida and the unknown reviewers for providing insightful and valuable comments and expert advice on text compression.

References

[1] S. Rein, C. Guhmann, F. Fitzek, Low complexity compression of short messages, Proceedings of the IEEE Data Compression Conference (DCC 2006), March 2006, pp. 123–132.
[2] S. Rein, C. Guhmann, F. Fitzek, Compression of short text on embedded systems, J. Comput. 1 (6) (September 2006) 1–10.
[3] Y. Hu, J.C. Zhang, F. Khan, Y. Li, Improving PPM algorithm using dictionaries, Proceedings of the IEEE Data Compression Conference (DCC 2011), March 2011, p. 459.
[4] 3GPP, Alphabets and language-specific information. TS 23.038, 3rd Generation Partnership Project (3GPP), http://www.3gpp.org/ftp/Specs/html-info/23038.htm (accessed on 3 January 2014).
[5] RFC989 - Privacy Enhancement for Internet Electronic Mail. Part I: Message Encipherment and Authentication Procedures.
[6] The Brown Corpus in plain text format, http://dingo.sbs.arizona.edu/hammond/ling696f-sp03/browncorpus.txt (accessed on 3 January 2014).
[7] The Canterbury Corpus, http://www.corpus.canterbury.ac.nz/ (accessed on 3 January 2014).
[8] NUS SMS Corpus, http://wing.comp.nus.edu.sg/SMSCorpus/ (accessed on 3 January 2014).
[9] 10,000 most common English words in order of frequency, as determined by n-gram frequency analysis of the Google's Trillion Word Corpus, https://github.com/first20hours/google-10000-english/ (accessed on 3 January 2014).
[10] M. Burrows, D.J. Wheeler, A Block Sorting Data Compression Algorithm, SRC Research Report 124, Digital Systems Research Center, Palo Alto, California, 1994.
[11] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, Springer, 2008.
[12] R. Franceschini, A. Mukherjee, A. Mukherjee, Data Compression using Encrypted Text, Proceedings of the Third Forum on Research and Technology. Advances on Digital Libraries, ADL96, 1996, pp. 130–138.
[13] M. Nelson, Star encoding, Dr. Dobbs J. (August 2002) 94–96 (Also see http://marknelson.us/2002/08/01/star-encoding-in-cpp/ (accessed on 3 January 2014)).
[14] A. Mukherjee, F. Awan, in: K. Sayood (Ed.), Text Compression, Chapter 10, Lossless Compression Algorithm, Academic Press, 2003.