Semi-Lossless Text Compression

Yair Kaufman and Shmuel T. Klein

Department of Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498

e-mail: {kaufmay,tomi}@cs.biu.ac.il

Abstract. A new notion, that of semi-lossless text compression, is introduced, and its applicability in various settings is investigated. First results suggest that it might be hard to exploit the additional redundancy of English texts, but the new methods could be useful in applications where the correct spelling is not important, such as in short emails, and the new notion raises some interesting research problems in several different areas of Computer Science.

Keywords: text compression, lossy, lossless compression

1 Introduction

One widespread partition when coming to classify data compression methods is into lossless and lossy methods. *Lossless* methods include usually those applied on text files or other data for which no loss of information can be tolerated, *lossy* techniques are generally applied to image files as well as to video and audio data, for which the overall knowledge a user might extract does not seem significantly reduced even if a part of the data is omitted.

Even though most lossy compression methods include some lossless techniques as one of their components, the research methods and goals of the corresponding communities are in fact quite different. While researchers in text compression are primarily concerned with good compression performance (in terms of speed and of space, both of the file to be compressed and of the RAM required by the method at hand), a major topic in image compression is finding a good tradeoff between the size of the compressed file and the ability of a human observer to find the differences between the original picture and its partially reconstructed copy. Many articles about image compression include side by side two pictures looking almost identical, the one labeled "original" and the other labeled "compressed". Obviously, the latter is rather the decompressed, reconstructed, image, the real compressed one consisting of a close to random sequence of zeros and ones, which would not yield any visual information when displayed as a raster file.

The basic idea behind lossy compression is thus the fact that even if not all of the available data is presented, the human brain can often make up for the missing parts and guess, at least partially, whatever has been omitted, so that overall one has the feeling that nothing has been lost. We try, in this paper, to transfer this paradigm

also into the framework of text compression, to which usually only lossless techniques have been applied.

A hint to the fact that strict losslessness might be relaxed can be found by anybody who tries to read a newspaper, and mostly succeeds in understanding all the required information in spite of occasional typing errors and other mistakes. It turns out that we are able to understand English text even if there are many more errors, as suggested by the following paragraph, which circulated recently on the Internet[1]

> Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are in; the olny iprmoetnt tihng is taht frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

If indeed it is true that under certain constraints the exact letter order can be altered without impairing our understanding of the information contained in English text, it follows that the order of the characters induced by English grammar and syntax may contain more redundancy than one thought so far, and eliminating this redundancy might yield improved compression. Being a hybrid of the two classes of compression methods mentioned above, we call the type of compression suggested by these ideas *semi-lossless*: the original text will not be fully reconstructed, just as a decompressed JPEG image is not identical to the original, and thereby the method will be lossy; on the other hand, again similarly to the decompressed image for which our eyes and brain fill in the omitted parts, here it is the knowledge of English that will enable the extraction of the full information of the original text, so that at least from the information point of view, if not from the physically stored file, the method can be considered as lossless.

A priori, the expexted gain from playing with the order of the characters within a word is not very large, as the average word in English is rather short (about 5 characters). The applicability of semi-lossless text compression might thus be restricted, most users preferring to get a clean text, even at the price of marginally lower compression. The new methods could therefore be useful in applications where the correct spelling is not important, such as in short emails or SMS notes sent between cellular phones, which already use some widely known shortcuts (that R acceptd by any 1). Moreover, the new notion raises many interesting research problems, some of which mentioned in the sequel, which may find applications in several different areas of Computer Science.

In the next section we suggest some approaches to semi-lossless text compression and discuss their usefulness as general data compressors. Section 3 then brings some preliminary experimental results, and we conclude in Section 4 with some possible extensions of this work.

[1]See, e.g., http://csunx4.bsc.edu/bmyers/language.htm, but there are dozens of pointers to this or similar phrases

2 Semi-lossless text compression techniques

Lossy text compression has already been suggested by Witten at al. in [9], which includes several quite amusing examples. We shall, however, concentrate on methods in which there is, at least a priori, no loss of information, and take the rule cited in the quotation in the introduction as a premise, namely, that if the first and last letters of the printed words are left in place, the remaining letters within each word can appear in any order.

This "law" is clearly not universal and relies on the assumption that the reader has a good knowledge of English. We are not concerned with checking the validity of this assumption, nor with suggesting alternative rules. This would rather fall into a domain investigated by psychologists, and the interested reader is referred to the vast literature dealing with several aspects of this subject, see, e.g., [2, 6] and the pointers appearing in their references. For our discussion it does not even really matter whether the given rule is true, or whether it should be reinforced (leaving the first, last and one or more additional letters in place) or could be relaxed (fixing only the first letter, or even none, allowing any permutation of any word). All we assume is that some rule exists according to which not all the characters of a text have to be restored to their original position for the text to be understandable. Such a rule will obviously depend on and vary according to language, potential readers and genre and type of the given text.

Taking therefore the quoted law (first and last letters fixed, the rest in any other position) as working assumption, it suggests the following generic compression and decompression algorithm:

Compression:
1. Process the words sequentially and if the current word is not special (number, proper name, etc.), do
2. keep first and last letters in place, but rearrange the others into "special" order;
3. apply some encoder on the rearranged text.

Decompression:
1. Decode the compressed words sequentially, and if the current word is not special, do
2. keep first and last letters in place; choose a random permutation of the other letters and send them to output.

Since the order of the characters (except the first and last of each word) is not restricted, it might be useful to choose a *special* order referred to in Step 2 of the compression, that will subsequently improve the encoding mentioned in Step 3. The reason for Step 2 of the decompression process is avoiding a constant bias introduced by the suggested partial order. It might be that seeing always the same permutations according to the special order chosen may interfere with our ability to recover the original word. Introducing the randomization restores for the reader the feeling of arbitrariness which, possibly, is necessary for correct decoding.

In the following sub-section, we explore some of the possibilities for choosing such a special order.

2.1 Choosing a special order of the characters

One possibility that comes to mind is arranging these letters in alphabetic order. The reason such a strategy is expected to improve compression is similar to the argument showing why the Burrows-Wheeler Transform (BWT) [1] actually works so well.

The BWT works on a string of length n and applies all the n cyclic rotations on it, yielding an $n \times n$ matrix which is then lexicographically sorted by rows. The first column of the sorted matrix is thus sorted, but BWT stores the *last* column of the matrix, which together with a pointer to the index of the original string in the matrix lets the file to be recovered. The last column is usually not sorted, but it corresponds to sorted contexts, and is therefore often very close to be sorted, which is why it is more compressible than the original string. The compression scheme based on BWT uses a move-to-front strategy to exploit this nearly sorted nature of the string to be compressed. Returning to our problem, if the characters in each word can be arranged alphabetically, this may similarly yield improved compression using move-to-front and/or run-length coding if the strings are long enough.

Another possibility would be to arrange the characters by frequency. The distribution of characters in English text is well-known, (see, e.g., [3]), and sorting the letters following the order `E`, `T`, `A`, `O`, `N`, `I`, `S`, etc., increases the probability of short displacements in move-to-front schemes. However, frequency of occurrence alone does not take the tight connections between certain characters into consideration.

A more precise rule would therefore be trying to group the characters based on the probabilities of a given letter to appear after another one. A strict approach gets quickly into loops, for example, `E` is most likely followed by `R`, which in turn has `E` as its most probable successor. A simple greedy algorithm would thus be:

1. Start with an arbitrary character, x;
2. While not all characters are processed
   - Choose, among remaining characters, the successor s of x with highest probability;
   - $x \longleftarrow s$

Following the probabilities in [3], one possible sequence this may yield is:

`A N D E R O U T H I S P L Y M B J - X C K - F - G - Q - V - W - Z.`

While the beginning of this sequence seems reasonable, there are some evident shortcomings: `P` as successor of `S` is only the 9th choice, because the eight preceding ones (in order: `T`, `E`, `I`, `S`, `O`, `A`, `U` and `H`) all appeared earlier. Towards the end, all the remaining potential successors have probability practically zero, indicated by the dashes, so the choice is arbitrary. Note also that choosing the successor with highest probability might push the second best choice too far away. The second most frequent successor of `A` is `T`, which appears only in seventh position after `A`.

These speculations lead to the following formulation of the problem: we seek an ordering of the letters maximizing the overall probability of the letter successions. More formally, let $\Sigma = \{x_1, \ldots, x_n\}$ be the alphabet, and let $P[x, y]$ denote the probability of character y appearing as successor of x; we look for a permutation

$\sigma : \Sigma \longrightarrow \Sigma$ of the n characters, such that

$$\prod_{i=1}^{n-1} P[\sigma(i), \sigma(i+1)] \tag{1}$$

is maximized. The following transformation shows that this is in fact an instance of the Minimum Traveling Salesperson Problem. Consider a full graph $G = (V, V \times V)$, with $V = \Sigma$, and define the weight $w(x, y)$ of an edge (x, y) as

$$w(x, y) = -\log P[x, y].$$

Finding a permutation maximizing (1) is then equivalent to finding a Hamiltonian path of minimum weight in G. Unfortunately, this is an NP-complete problem, and since in our case, there is no reason to assume that the triangle inequality holds for the weights, it might even be hard to find a good approximation.

We now turn to the more technical details of choosing a specific compression scheme.

2.2 Choosing the compression technique

A simple statistical encoder, such as Huffman or arithmetic coding, applied independently to the individual characters will, of course, not yield any additional compression at all. The set of encoded characters remains the same, only their order is altered. To be able to take advantage of the partial reordering, a method is needed that takes previous characters into account.

A simple example would be run-length encoding, which is not likely to be useful. Run-length coding is widely used for images or fax-transmission, but in natural language text there are hardly any repeated strings of length longer than 2 (in German, there are some rare examples of runs of length 3, such as in `Schifffracht`). In our case, where the internal characters appear in sorted order, the lengths of runs are still limited by the number of times a given letter appears within a word. But the average word length, in English, is only about 5, so that no significant runs may be expected (German provides here again an extreme case: there is a street in Vienna named `Abrahamasantaclaragasse`, which would give a run of 9 `a`'s).

We may expect better performance when using Huffman or arithmetic coding in connection with a Markov model of order $k \geq 1$, meaning that each character is encoded as a function of the k characters preceding it. Though even natural text is well compressed by such a model as it captures many of its characteristic features (`q` followed by `u`, high probability for `e` following `th`, etc.), having identical characters grouped together may even cause better compression. However, the additional space requirements of higher order Markov models may be prohibitive.

Adaptive methods like Lempel-Ziv variants seem at first sight not applicable. In an adaptive encoding, the current item to be encoded relies on previously seen text, and if the item is not reliably restored, a subsequent pointer to it may give wrong results. Consider, for example, the string

$$\cdots \texttt{a b c x y z c b a d e f w t w c e d f a v} \cdots$$

to be encoded by LZSS [4], and suppose that the whole string consists of internal characters (not the first or last in a word). The string `cba` can then be replaced by

a pointer to the preceding `abc`, and `edfa` could point to `adef`, so that the modified LZSS encoding would be

$$\cdots \texttt{a b c x y z } (6,3) \texttt{ d e f w t w c } (8,4) \texttt{ v } \cdots .$$

But while the first pointer (6,3) would be decoded to `abc`, as expected, the second pointer (8,4) would now refer to the substring `cdef`, which is *not* a permutation of the original `edfa`. Note that the problem here is caused by the overlap between a substring, `cba`, that is replaced by an (*offset*, *length*) pointer, and a substring, `adef`, which is the target of such a pointer. In the absence of such overlaps, the encoding scheme works correctly.

One strategy to avoid the problem would thus be to forbid such overlaps, but this would affect compression efficiency. Another possibility is to adapt LZSS to work in this case, by keeping a copy of the currently decoded text, and search in it, rather than in the original text processed so far, for earlier occurrences of the current string to be encoded or its permutations. Returning to the example above, after having encoded `cba`, the processed string would look as

$$\cdots \texttt{a b c x y z a b c} \mid \texttt{d e f w t w c e d f a v} \cdots$$

where the vertical bar indicates the current position, and to its left appears the reconstructed, rather than the original, text. While the bar now moves further to the right, the string `edfa` cannot be encoded as before. However, in this example, even a better substitution is possible, replacing `wcedf` by a pointer to `cdefw`, so that the encoded string finally looks as

$$\cdots \texttt{a b c x y z } (6,3) \texttt{ d e f w t } (6,5) \texttt{ a v } \cdots .$$

In fact, a correct algorithm based on LZSS is even more involved. Fast implementations of LZSS, like LZRW1 [8] or Microsoft's DoubleSpace [7] find the recurring strings by locating, using hashing, a previous occurrence of the character pair following the current position, and then extending the strings as far as possible by checking if the subsequent characters coincide. In our case, such a greedy approach may fail, e.g., for the string

$$\cdots \texttt{x y z t a b c d e f g} \cdots \texttt{x z y t a b e d c h k} \cdots .$$

The second occurrence of `ab` would point to the first one, but trying to extend the strings would fail in the first two attempts, `abe` and `abed` not matching `abc` and `abcd`, respectively, and only the third attempt would succeed, with `abedc` matching `abcde` modulo the reordering. Moreover, word boundaries have to be taken into account because of the constraint that first and last letters have to remain in place. The processing must therefore be by a combination of trying to extend partial matches by entire words and, once this fails, trying to match prefixes of the last word dealt with, proceeding backwards from the longest to the shorter ones. In the above example the word `xzyt` is first matched to `xyzt`, trying then to match `abedchk` to `abcdefg` fails, so we try to backtrack. `abedch` does not match `abcdef`, but `abedc` does match `abcde`, which gives the string `xzyt abedc` as required match.

Similar problems to those of LZSS would arise in LZ78 variants like LZW [5]. Instead of pointing to earlier strings in the already processed text, the compressed file consists of a series of pointers to an external dictionary, which is built on the fly. Here again, relaxing the rules and letting a pointer refer not necessarily to the string to be replaced, but possibly to any of its permutations, may yield some savings: the overall number of strings is reduced, implying that more good strings can be stored, or that the necessary pointers can be shorter. But as above, decoding may be erroneous, because the strings stored by LZW are overlapping, specifically, the last character of the nth stored string is also the first of the $n+1$st.

The problem may be more severe in this case, because eliminating one of the strings stored in an LZW dictionary will affect all the subsequent entries and therefore change all subsequent pointers, whereas for LZSS, all the changes are locally restricted.

2.3 Combining character ordering and compression technique

A different approach than trying to adapt Lempel-Ziv type methods would be to restrict ourselves to dealing with bigrams, trigrams, or generally, any k-grams with $k > 1$. Each word is considered on its own, and decomposed into a sequence of such consecutive k-grams, leaving, as before, the first and last letters in place. Special care is needed to deal with the last k-gram in this sequence within a word, which might require a smaller k. Then each k-gram is mapped to a representative, in a predetermined order (alphabetic, `ETAONI`, `ANDERO`, etc.). Finally, the items obtained by this decomposition are Huffman coded. Since the number of different k-grams is reduced from $|\Sigma|^k$ to $\binom{|\Sigma|}{k}$, a savings of about 50% for $k = 2$, and more for higher k, the average Huffman codeword lengths are expected to be lower. Moreover, the overhead of storing the different k-grams is also reduced.

An alternative would be to process the k-grams sequentially, without taking word boundaries into account. Each k-gram would again be mapped to a reordered one, but flag-bits would be used to indicate if there has been a reordering and which one. For bigrams, a single bit suffices to indicate whether to switch a pair, and the bit is needed only for those pairs following or preceding a space.

Block sorting using the BWT could also be adapted to our case. As mentioned earlier, the last column, which is the one stored by the algorithm, is almost sorted. Suppose we have a sequence of the form `A`, `A`, `A`, `B`, `A` $\cdots$ in this column. If we can change the order of the characters, we might want to remove the `B` from within the sequence of `A`s. Work on a general algorithm based on this idea is ongoing.

3 Experimental results

The first text chosen as testbed for the above semi-lossless algorithms consists of about 3MB of the AP newswire files from the TREC collection. In addition, the methods were applied to Mark Twain's *Tom Sawyer* taken from the Gutenberg Project. To avoid a bias introduced by punctuation and other signs, all non-alphabetic characters, except the space, have been removed, and all the others have been changed to upper case, giving an alphabet of size 27.

Table 1 summarizes some of the results. The first column gives the size of the raw files, the second after having applied simple Huffman coding on the individual letters. All compression figures are given in bits per character (bpc). The next columns deal with bigrams and trigrams, first in a standard fragmentation of the text into bi- or trigrams, then using the reordering for those k-grams that can be changed. For the bigrams the variant with the flag-bit has been applied, for the trigrams, triples including the first or last letter of a word have not been reordered. The figures include the overhead of storing the bi- or trigrams.

	size	Huffman	bigrams		trigrams	
			standard	ordered	standard	ordered
AP	2.57 Mb	4.148	3.791	3.707	3.529	3.437
Tom Sawyer	361 Kb	4.111	3.707	3.687	3.549	3.485

As can be seen, there is a slight improvement, though not a significant one. In fact, even with better parsing strategies than the simple one we used, one should not expect large savings for English text: the average word length being less than 5, and the two corner letters being fixed, the reordering will affect on the average less than 3 letters. However, with schemes going beyond word boundaries, like LZSS, or for other languages and other reordering rules, better results might be expected.

Note that there are far better compression schemes: applying Huffman coding on the basis of words, rather than characters, yields, for AP, 2.136 bpc, and if the internal letters of the words are reordered, 2.135 bpc, saving less than 0.05 percent. But such a scheme requires a large overhead for the storage of the Huffman tree, and can only be justified if the set of different words is stored anyway, e.g., as the dictionary in an Information Retrieval system.

4 Conclusions and future work

The main contribution of this paper is thus not the presentation of some novel compression technique, but rather the introduction of the notion of semi-lossless text compression and the ensuing research problems it raises in compression, pattern matching, computational linguistics and possibly other related areas. We have briefly explored how some of the known compression methods could be adapted to take advantage of the relaxed constraints and work currently on implementing some of the advanced methods.

Much work is still to be done. Here is a partial list of topics one might want to deal with:

- One could try to devise new methods that do not rely on adapting existing ones, but may possibly be totally different and specially adapted to our case.
- Different languages may suggest other rules. In *French*, grammatical suffixes are more abundant and often one or more of the last letters of a word are not even pronounced. Perhaps the rule of keeping specifically the last letter in place is then not adequate? *German* has the ability of concatenating several words

into a single one; should the rule then be extended to fix also letters at sub-word boundaries, and how could these boundaries be detected? The average length of a word in *Finnish* is much longer than in English and double letters are more frequent.

- One could adapt ideas from other languages to English. For instance, *Hebrew* is generally written without vowels. This gives a large number of possible interpretations for each word, most of which are grammatically incorrect, but on the average, every word has four possible correct readings. Nevertheless, a native speaker has generally no trouble to pick the right choice quickly enough to read fluently, partly because certain consonants may act as vowels. It would not be reasonable to strip all the vowels from English texts (thgh ths wld gv gd cmprssn!), but perhaps one can devise rules to get rid of most of them, as we do anyway in speed-writing or when sending short electronic notes by computer or on cellular phones.

- Semi-lossless compression is not necessarily restricted to keeping a permutation of the original characters. When typing on cellular or regular phones, each key is assigned to several characters and the requested one is reached by repeatedly pressing the same key. It may be that the sets assigned to each key can be chosen in such a way that pressing only once, and thereby sending a representative of a small set, can still result in a text that is understandable. The size of Σ would be reduced, so one may save space, but also the time necessary to type a message will be greatly shortened.

Another short note many web-user got lately in their mail claimed that English spelling will shortly be simplified[2]. While this was meant as a joke, the idea is another nice example of how semi-lossless techniques could be implemented. The text suggested a five year plan during which many old spelling rules would be gradually abolished or modified, until

> after zis fifz yer, ve vil hav a reli sensibl riten styl. zer vil be no mor trubls or difikultis and evrivun vil find it ezi tu understand ech ozer.

While most of us will easily decipher the quote, note that its length (148 characters) is 14% shorter than its correctly spelled equivalent (172 characters), and the same 14% gain is also obtained if each of the messages is Huffman encoded (600 instead of 694 bits).

References

[1] Burrows M., Wheeler D.J., A block-sorting lossless data compression algorithm, Technical Report SRC 124, Digital Systems Research Center, Palo Alto, CA (1994).

[2] Friedmann N., Gvion A., Letter position dyslexia, *Cognitive Neuropsychology* **18**(8) (2001) 673–696.

[2] http://www.bluegum.com/Humour/Assorted/easier-english.htm

[3] Konheim A.G., *Cryptography, A Primer,* John Wiley & Sons, New York (1981).

[4] Storer J.A., Szymanski, T.G., Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.

[5] Welch T.A., A technique for high performance data compression, *IEEE Computer,* **17** (1984) 8–19.

[6] Whitney C., How the brain encodes the order of letters in a printed word: The SERIOL model and selective literature review, *Psychonomic Bulletin & Review* **8**(2) (2001) 221–245.

[7] Whiting D.L., George G.A., Ivey G.E., Data Compression Apparatus and Method, U.S. Patent 5,126,739 (1992).

[8] Williams R.N., An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah (1991) 362–371.

[9] Witten I.H., Bell T.C., Moffat A., Nevill-Manning C.G., Smith T.C., Thimbleby H., Semantic and generative models for lossy text compression, *The Computer Journal* **37**(2) (1994) 83–87.