RJ 10260 (A0210-041) October 24, 2002
Computer Science

IBM Research Report

Efficient Short String Compression with the Burrows-Wheeler Transform

Cornel Constantinescu, J. Q. Trelewicz, Ron Arps
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

EFFICIENT SHORT STRING COMPRESSION WITH THE BURROWS-WHEELER TRANSFORM

Cornel Constantinescu, J. Q. Trelewicz, and Ron Arps

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120 USA
e-mail: cornel@us.ibm.com, trelewicz@us.ibm.com, arps@almaden.ibm.com

ABSTRACT

There are applications (such as Internet search engines) where short textual strings, for example abstracts or pieces of Web pages, need to be compressed independently of each other. The usual adaptive compression algorithms perform poorly on these short strings due to the lack of necessary data to learn.

In this manuscript, we introduce a compression algorithm targeting short text strings; e.g., containing a few hundred symbols. The algorithm is based on the following findings. Applying the move-to-front transform (MTFT) after the Burrows-Wheeler transform (BWT) brings the short textual strings to a "normalized form" where the distribution of the resulting "ranks" has a shape similar over the set of natural language strings in a given language. This facilitates the use of a static coding method with few variations, which we call `shortBWT`, where no on-line learning is needed, to encode the ranks. Finally, for short strings, `shortBWT` runs very fast because the strings fit into the cache of most current computers.

1. INTRODUCTION

Compression of symbolic information constructed according to a grammar, such as textual strings, has long been a topic of interest in computer science. The Burrows Wheeler transform (BWT), combined with some form of the move-to-front transform (MTFT), has received significant attention in recent years. Applying the MTFT after the BWT brings textual strings to a "normalized form", utilizing contextual information inherent in language grammars. Alone, these two transforms do not compress the data but utilize contextual redundancy to facilitate entropy coding. The BWT and MTFT are invertible, still permitting subsequent lossless compression. The "ranks", output from the MTFT, are then entropy-encoded to achieve the needed compression.

In applications such as Internet search engines, short textual strings need to be compressed independently of each other. These strings may comprise titles or abstracts of the pages. The usual adaptive compression algorithms give relatively low relative compression gain (rCG) on these short strings due to the lack of necessary data to learn. rCG is defined as the relative decrease in size of the string, as a result of compression. Furthermore, the application is extremely sensitive to speed of computation, since compression and decompression must occur in real time. In [1], it is shown that the speed of convergence of BWT-type compression is faster than, e.g., that of Lempel-Ziv-type algorithms, suggesting the suitability of BWT-type algorithms for this type of application.

Much of the BWT and MTFT literature has debated the suitability of one algorithm configuration over another, justifying these results against standardized test suites of large files. However, there has been little activity in short string compression, where computation speed is critical.

In this paper we introduce a notation for the BWT and MTFT to facilitate the subsequent discussion. We give an overview of the recent literature concerning these transforms applied to the compression of symbolic grammatical data. We then develop our algorithm, `ShortBWT`, for short string compression, comparing its compression performance to that of several popular compression algorithms.

2. UNDERSTANDING THE TRANSFORMS

This section gives an overview of the transforms, developing a notation. An illustrative example is provided in Sec. 2.4 to aid the reader in understanding the operation of the transforms.

2.1. Burrows-Wheeler Transform

Let finite, well-ordered, symbol alphabet A be given, with length $|A|$ and lexicographical ordering $>$ (extending naturally to $<$, $\leq$, and $\geq$). Define $\$ \notin A$ the sentinel symbol such that $\$ > a \; \forall a \in A$. Consider S a finite string of symbols in A, and let $|S|$ be the length of S in symbols. The BWT is denoted by T_B, which permutes S. The inclusion

of the symbol \$ facilitates the inversion of T_B in this configuration. Some other configurations of the BWT do not employ the sentinel, using other methods for inversion.

The BWT is used to group *locally-frequent* symbols in a string. The transform is invertible, thus suitable for lossless compression of text information. All substrings of S, each terminated by \$, are sorted lexicographically by the *second* symbol in the symbol-order-reversed substring. The value of $T_B(S)$ is the first symbol of each substring, in the sorted substring order. For example, if $S =$ `banana`, $T_B(S) =$ `nnaaab`, with an implicit \$ at the end of $T_B(S)$.

The BWT can be seen as sorting the symbols of the textual string according to their symbol contexts, thus clustering in $T_B(S)$ the symbols that appear in the same context. These substrings, or suffixes, may be ordered through construction of a suffix array [2], essentially a sorted suffix tree. Using conventional algorithms, the suffix tree may be constructed in $O(|S| \log |A|)$ time in $O(|S|)$ space. Much of the recent interest in the BWT is based on this characteristic, making it well-suited for use in natural language string compression.

The BWT by itself does not provide compression, but when combined with a transform like the MTFT and an entropy code, compression of the text information may be achieved. Efficient implementations of T_B are described in, for example, [3].

2.2. Move-to-Front Transform

The MTFT is denoted by T_M, taking a string (such as $T_B(S)$) to $\mathcal{R}$, the set of finite sequences in $\mathbb{N}$. These sequences are called *ranks*, and are used to compress S, by providing position and context information for suffixes of S.

Any localized region of $T_B(S)$ is likely to contain a large number of a few distinct symbols. The overall effect is that the probability that given symbol will occur at a given index j in $T_B(S)$ higher if that symbol occurs near index j in $T_B(S)$. This property is exactly that needed for effective compression by T_M, which encodes an instance of a symbol by the count of distinct symbols encountered since the most recent occurrence of that symbol. When T_M is applied to $T_B(S)$, the output is ideally dominated by low numbers, which can be efficiently encoded with a Huffman or arithmetic coder.

The MTFT operates in order on the elements of $\kappa = T_B(S)$, giving $t = T_M \circ T_B(S)$ with sequence elements $t_1 t_2 \ldots t_{|S|}$. In what follows, $P^j = \{p^{j,k}\}_{k=1}^{|A|}$ is a permutation of A, with P^1 defined as A and

$$\begin{aligned} t_j &= k : p^{j,k} = \kappa_j \\ P^{j+1} &= (\kappa_j, \kappa_1, \ldots, \kappa_{j-1}, \kappa_{j+1}, \ldots, \kappa_{|A|}) \end{aligned}$$

Thus, as T_M steps through the string, it moves the current symbol in the string to the front of the permuted alphabet and outputs as the rank the previous position of the symbol in the alphabet. The MTFT can be seen as associating smaller integers with frequently-occurring symbols in S.

Although T_M does not compress S, it can help subsequent entropy coding to reduce redundancy. When T_M follows T_B, locally-frequent symbols for every context of the given string are grouped. Even though T_M is language independent, a language-sensitive initial ordering of A can help to improve the rCG by reducing the magnitude of t_j by placing frequently-occurring symbols near the beginning of A. In [4], a non-lexicographical ordering of A is recommended, which affects $T_M \circ T_B$ and thus impacts rCG. If the language of the text is known in advance, A may be ordered according to statistics for the language.

A number of improvements on T_M for specific applications are discussed in the literature. In [5], a dictionary of common words in the target language is used, where these words are replaced with alternate strings of symbols, which in their application provides improvements in the rCG of up to 20% over `bzip2` (a freeware implementation of a BWT-based coder using adaptive arithmetic entropy coding, adapted from [3]). In [6], Inversion Coding replaces T_M, which is shown to provide better rCG than $T_M \circ T_B$ for *large* files, but comparable results for small files.

Some implementations avoid T_M altogether. In [7], symbols in are encoded directly, without ranks. This is shown to provide higher rCG on large files, but is significantly slower than T_M. Our application must run in real time, so that runtime speed must be traded against rCG. Also, the gains in rCG obtained coding symbols directly are not significantly high (on the order of 2%) to justify the additional computational complexity for our application.

2.3. Huffman coding

It is shown in [8] that the output of T_B is approximately memoryless and piecewise stationary, making it appropriate for Huffman entropy coding [9]. As can be seen in the compression comparison of Fig. 2, using self Huffman codebooks (i.e., separate Huffman codebooks trained on each compressed string) gives the best compression. However, in this case the Huffman codebook must be encoded and sent to the decoder, significantly impacting both rCG and speed for our short-string application.

Alternatively, the Huffman codebook can be approximated to simplify its encoding (without significantly affecting the overall code lengths). It should be noted that this method is robust, while providing some adaptivity. Alternatively, the encoder may employ a set of predetermined Huffman codebooks, sending to the decoder the index of the appropriate codebook. The codebook may be chosen by, for example, a metric such as that mentioned in Sec. 2.6.

2.4. Illustrative Example

The following small example is discussed to illustrate how $T_M \circ T_B$ can lead to compression of text. Consider the effect on a single symbol in the common English word "the", and assume that the input string is long enough to have several instances of this word. In $T_B(S)$, all substrings beginning with "the" will be grouped together. Because T_B operates on the symbol-order-reversed string (e.g., where "the" becomes "eht"), a large proportion of them are likely to end in "t". The reason for this is that English has many words beginning in "t". One region of $T_B(S)$ will contain a large number of suffixes beginning in "t", intermingled with other symbols that can precede "the". The same argument can be applied to all symbols in all words, so any localized region of the transformed string is likely to contain a large number of a few distinct symbols.

2.5. Alternative Implementations

It has been shown (e.g., [10]) that a natural language can be modeled by an exponential distribution in language features, such as w. We denote the language parameter q in the distribution; i.e., the probability that two suffixes in the language are identical in the first w symbols and different in the $w-1$st is approximately qe^{-qw} for large w. This parameter q can be sent in the compression header, giving a similar rCG advantage as the Huffman codebook approximation method, discussed in Sec. 2.3 above.

2.6. Screening Strings

Because this short string application is extremely sensitive to computation speed, it is important to be able to estimate whether a short string will compress with `ShortBWT`. However, performing this screening must not significantly slow the compression.

In [11] we developed a computational method for `ShortBWT` that compresses the string iteratively. The algorithm builds a two-dimensional function, based on the alphabet A and the truncated suffixes of S, iterating on the maximum truncated suffix length w. The two-dimensional function is used to compute a metric, which is an approximation of the expected rCG. Although the details are beyond the scope of this paper, our previous work shows that the metric converges quickly to the actual rCG for natural-language strings. The building of the two-dimensional function also computes the compressed output directly. Thus, the metric can be used to abort the computation quickly if little or no compression is indicated.

3. DETAILS

For short textual strings, of few hundreds symbols, it is illustrated in Fig. 1 that the distribution of the symbols frequencies in $T_M \circ T_B(S)$ is very similar for different strings in a given language (in this case, English). Thus, a static Huffman coder can efficiently encode the transformed string.

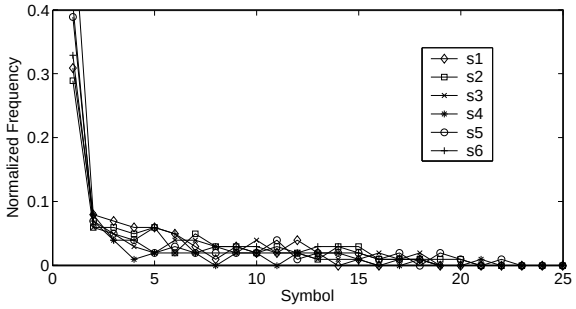

Figure 1: Symbol frequencies after $T_B \circ T_M$.

To build the Huffman code for use over a very large set of short strings, a relatively small set of strings was used as a training set. These strings were individually transformed ($T_B \circ T_M$) and the frequency of occurrence of each rank was accumulated. To waste the least possible code space, a frequency of 1 was assigned to each of the 256 possible ranks that did not occur with the training set, and all the other rank frequencies were multiplied by a constant (50 in our experiment). By doing this frequency assignment, the alphabet does not need to be explicitly encoded. Based on these frequencies we chose to generate a canonical Huffman code, where the Huffman algorithm is used to get the lengths of the codewords, but the codewords for each length are consecutive binary numbers [12]. The canonical Huffman code has several well-known advantages: fast decoding and minimal memory required to hold the codebook. These advantages are especially useful when the codebook has to be sent to the decoder.

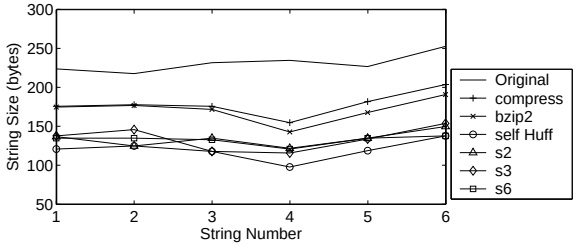

Figure 2: The compressed sizes on six realistic sample textual strings.

In Fig. 2 we compare `ShortBWT` (using a static Huffman coder) with available adaptive coding algorithms: UNIX

`compress` and `bzip2`. The "self Huff" configuration uses `ShortBWT` with a self Huffman table for each string. Configurations "s2", "s3" and "s6" use `ShortBWT` with the static self Huffman tables for strings s2, s3, and s6, respectively, to encode all of the other strings. It can be seen from the figure that using any of the static Huffman tables (s2, s3, or s6), the compressed sizes are smaller than for the alternative, `bzip2` or `compress`, algorithms. Also, the choice of which Huffman table to use is not as critical; this is because the frequency distribution of the symbols in any of the transformed strings are very similar, as seen in Fig. 1.

In Fig. 3, we show the result of compressing 1000 short textual strings (with lengths on the order of a few hundred symbols) with `ShortBWT` and some popular, conventional text compression algorithms. It can be seen from the figure that `ShortBWT` provides about 30% rCG improvement over the comparison algorithms.

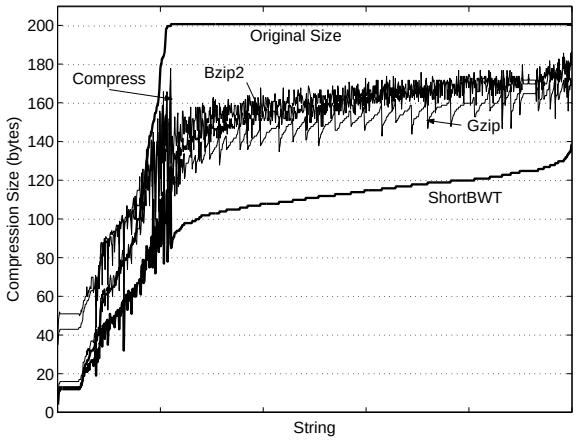

Figure 3: Compression comparison of `ShortBWT`, `compress`, `gzip`, and `bzip2` on 1000 short textual strings.

4. CONCLUSIONS AND FUTURE WORK

We have introduced an algorithm for compression of short strings, with applications in Internet search engines and other indexing operations. Our algorithm has been shown to perform better than other, popular algorithms used for compression of textual strings. This algorithm can be implemented using iterative technology that we introduced previously, facilitating screening of the strings for expected rCG.

We are also exploring improvements to the algorithm. For example, if the distribution of zeros is very skewed the algorithm can use arithmetic coding to encode events as zero or non-zero, using Huffman coding to encode only the non zero symbols.

5. REFERENCES

[1] M Effros, "Universal lossless source coding with the Burrows Wheeler transform," in *Proc. DCC'99 Data Compr. Conf.*, Snowbird, UT, USA, 29-31 March 1999, pp. 178–187.

[2] U Manber and G Myers, "Suffix arrays: a new method of on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, Oct. 1993.

[3] P M Fenwick, "The Burrows-Wheeler transform for block sorting text compression: principles and improvements," *Comput. J.*, vol. 39, no. 9, pp. 731–740, 1996.

[4] B Chapin and S R Tate, "Higher compression from the Burrows Wheeler transform by modified sorting," in *Proc. DCC'98 Data Compr. Conf.*, Snowbird, UT, USA, 30 March - 1 April 1998, p. 532.

[5] H Kruse and A Mukherjee, "Improving text compression ratios with the Burrows Wheeler transform," in *Proc. DCC'99 Data Compr. Conf.*, Snowbird, UT, USA, 29-31 March 1999, p. 536.

[6] Z Arnavut, "Move to front and inversion coding," in *Proc. DCC 2000 Data Compr. Conf.*, Snowbird, UT, USA, 28-30 March 2000, pp. 193–202.

[7] A I Wirth and A Moffat, "Can we do without ranks in Burrows Wheeler transform compression?," in *Proc. DCC 2001 Data Compr. Conf.*, Snowbird, UT, USA, 27-29 March 2001, pp. 419–428.

[8] K Visweswariah, S Kulkarni, and S Verdu, "Output distribution of the Burrows Wheeler transform," in *Proc. 2000 IEEE Int'l Sym. on Information Thy.*, Sorrento, Italy, 25-30 June 2000, p. 53.

[9] T M Cover and J A Thomas, *Elements of Information Theory*, John Wiley and Sons, Inc., New York, 1991.

[10] S F Chen, K Seymore, and R Rosenfeld, "Topic adaptation for language modeling using unnormalized exponential models," in *Proc. 1998 IEEE ICASSP, II*, Seattle, WA, USA, 12-15 May 1998, pp. 681–684.

[11] J Q Trelewicz, Cornel Constantinescu, and Ron Arps, "A metric for compression with the Burrows-Wheeler and move-to-front transforms," *to appear in Int. Jnl. of Comp. and Num. Anal. and Apps.*, 2002.

[12] D Hirschberg and D Lelewer, "Efficient decoding of prefix codes," *Comm. ACM*, vol. 33, no. 4, pp. 449–459, Apr. 1990.