Grammar Precompression Speeds Up Burrows–Wheeler Compression*

Juha Kärkkäinen, Pekka Mikkola, and Dominik Kempa

Department of Computer Science, University of Helsinki, Finland
{juha.karkkainen,pekka.mikkola,dominik.kempa}@cs.helsinki.fi

Abstract. Text compression algorithms based on the Burrows–Wheeler transform (BWT) typically achieve a good compression ratio but are slow compared to Lempel–Ziv type compression algorithms. The main culprit is the time needed to compute the BWT during compression and its inverse during decompression. We propose to speed up BWT-based compression by performing a grammar-based precompression before the transform. The idea is to reduce the amount of data that BWT and its inverse have to process. We have developed a very fast grammar precompressor using pair replacement. Experiments show a substantial speed up in practice without a significant effect on compression ratio.

1 Introduction

Burrows–Wheeler compression is a popular text compression method consisting of two main phases, the Burrows–Wheeler transform (BWT), which produces a permutation of the text, and entropy coding, which performs the actual compression. There are very fast entropy coders [1,2], but the computation of the BWT during compression and the inverse BWT during decompression take too much time for Burrows–Wheeler compression to compete against the fastest text compression methods (usually based on Lempel-Ziv compression) in speed [10].

One way to speed up the computation of BWT and its inverse is to divide the text into smaller blocks and compress each block separately. However, this can impair compression ratio as the compressor cannot take advantage of redundancies that cross the block boundaries [7].

In this paper, we propose a method for speeding up Burrows–Wheeler compression and decompression without a significant effect on the compression ratio. The basic idea is to perform *grammar compression* of the text *before* the BWT; this is called *precompression*. The goal is to reduce the size of the text, which naturally reduces the time for the BWT and its inverse.

Grammar compression is similar to Lempel–Ziv compression in that it is based on replacing repeated substrings with references, but it is better suited for precompression because of the consistency of references. In Lempel–Ziv compression, each reference is usually unique, while in grammar compression, different

* Supported by Academy of Finland grant 118653 (ALGODAN).

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 330–335, 2012.

occurrences of the same substring are replaced by the same reference, a new non-terminal symbol. Thus grammar compression leaves the door open for further compression involving the new symbols, which makes it ideal for precompression. Indeed, some grammar compression algorithms operate in iterations, with each iteration compressing the text further.

We have developed a grammar precompressor based on replacing frequent pairs of symbols with non-terminals. It is very fast because it makes only a few sequential passes over the text during compression, and only one pass during decompression. Experiments show that the time spent in precompression is usually much less than the time gained in faster BWT. A similar speed up is obtained for decompression even with the recent improvements in the speed of inverse BWT [8]. The effect of the precompression on the compressibility of the data is insignificant according to our experiments.

Precompression is not a new idea, but usually the goal is to improve compression ratio for special types of data. For example, there has been a lot of work on grammar precompression of natural language texts; see [12] and the references therein. Another type of universal precompressor, replacing long repeats by Lempel–Ziv style references, is described in [3].

Replacing frequent pairs has been used in standalone compressors. Re-Pair [9] is perhaps the best-known of them but it is too slow to be used as a precompressor. The compressors proposed in [11] and in [4] are similar to ours but not identical (see Section 2 for details). Similar techniques for pairs of words instead of symbols are used in [6].

2 Grammar Precompression

In grammar-based text compression, the goal is to construct a small context-free grammar that generates the text to be encoded and no other strings. Finding the smallest grammar is a hard problem even to approximate well [5]. However, in precompression, fast execution is more important than the best possible compression rate.

Our grammar compression was inspired by Re-Pair [9], which repeatedly replaces the most frequent pair of symbols until no pair occurs more than once. Using sophisticated data structures, the whole procedure runs in linear time, but it is too slow for precompression.

To speed it up, we repeat the following a few times (see Fig. 1 for an example):

1. Compute the frequencies of symbol pairs by scanning the text.
2. Choose a set of frequent pairs that cannot overlap (see below).
3. For each chosen pair AB, add the rule $X \to AB$, where X is a new non-terminal symbol.
4. Replace all occurrences of all chosen pairs with the corresponding non-terminal symbols using a single sequential pass over the text.

Decompression is performed by computing the full expansions of all rules and replacing them with a single pass over the text. The speed of the precompressor

Text	Rules added
`singing_do_wah_diddy_diddy_dum_diddy_do`	$A \to$ `_d`, $B \to$ `id`, $C \to$ `in`
`s`C`g`C`g`A`o_wah`AB`dy`AB`dy`A`um`AB`dy`A`o`	$D \to AB$, $E \to$ `dy`, $F \to A$`o`, $G \to C$`g`
`s`GGF`_wah`$DEDEA$`um`DEF	

Expanded rules: $A \to$ `_d`, $B \to$ `id`, $C \to$ `in`, $D \to$ `_did`, $E \to$ `dy`, $F \to$ `_do`, $G \to$ `ing`

Fig. 1. Example of the grammar precompression with two rounds and the expansion of rules during decompression

is based on the fact that the sequential passes over the text are very fast in practice. Cannane and Williams [4] proposed a similar algorithm as a standalone compressor, but they choose the pairs differently.

Choosing the pairs. To maximize the compression, we want to choose as many pairs as possible in each round. A simple option would be to choose all pairs with a frequency above a threshold. However, if occurrences of two pairs overlap, we cannot replace both occurrences. This can lead to inoptimal encoding as illustrated in the following example.

Example 1. Let $T =$ `abcabca` be the text. The pairs `ab`, `bc` and `ca` occur twice each, so we create the rules $X \to$ `ab`, $Y \to$ `bc` and $Z \to$ `ca`. A greedy replacement produces the text XZY`a`, and we ended up using each rule just once. No further compression of the text is possible, since each symbol occurs just once.

On the other hand, if we choose just the pair `ab` and the rule $X \to$ `ab` at first, the replacement produces the text X`c`X`ca`. Then a second round with the rule $W \to X$`c` results the text WW`a`. Thus, instead of three rules and a text of length four, we have two rules and a text of length three.

Cannane and Williams used an extra scan of the text to estimate the pair frequencies when taking overlaps into account. Our approach is to choose, in each round, only pairs that cannot overlap. Formally, pairs A_1B_1 and A_2B_2 can overlap if and only if $A_1 = B_2$ or $B_1 = A_2$. This was already proposed by Manber [11] (but doing only one round and no further compression). Manber used an iterated local search heuristic to find a good set of pairs. We use a simpler approach that scans the pairs in a descending order of frequency and selects greedily each pair that cannot overlap any already selected pair until the frequencies drop too low.

Encoding symbols. Re-Pair uses zero-order entropy coding for the text and a sophisticated method for encoding the rules. In our case, the entropy coding happens later, so we simply append the rules to the text. One potential problem, though, is the size of the alphabet. Most BWT implementations are specialized for sequences of bytes limiting the alphabet size to 256, but with the addition of new non-terminal symbols the alphabet can grow bigger. We could simply add only as many rules as there are unused byte values, as Manber does [11], but with the multiple rounds of our precompressor, this is not sufficient for us.

We address this problem by encoding frequent symbols with a single byte and rare symbols with two bytes. Let $\mathcal{B} = \{0, 1, \ldots, 255\}$ be the byte alphabet. We

Table 1. Files used in the experiments. The files are from (L) the Large Text Compression Benchmark (`http://mattmahoney.net/dc/text.html`), (S) the Pizza & Chili standard corpus (`http://pizzachili.dcc.uchile.cl/texts.html`), and (R) the Pizza & Chili repetitive corpus (`http://pizzachili.dcc.uchile.cl/repcorpus.html`). n = text length, σ = alphabet size.

Name	σ	$n/2^{20}$	Source	Description
kernel	160	247	R	36 versions of Linux Kernel sources
enwik9	206	954	L	Wikipedia XML
dna	16	386	S	part of Human genome

divide $\mathcal{B}$ into two disjoint sets $\mathcal{B}_1$ and $\mathcal{B}_2$. A symbol can be encoded either by a single byte value from $\mathcal{B}_1$ or by a pair of byte values from $\mathcal{B}_2$. This encoding supports alphabet sizes up to $|\mathcal{B}_1| + |\mathcal{B}_2|^2$.

3 Experimental Results

We implemented the precompressor described in Section 2 and performed experiments to test three hypotheses:

1. The grammar precompression improves the compression time.
2. The grammar precompression improves the *de*compression time.
3. The grammar precompression does not hurt the final compressibility of the data significantly.

In the compressibility experiments, we use a very good (but slow) entropy coder, which is competitive with the best compressors of any type (see Table 2). This way any harmful effects of the precompression are exposed.[1] Otherwise, we have excluded results on entropy coders. Such results would not be representative as there are a wide variety of entropy coders and we are in the process of developing our own. Excluding the entropy coder times is not a serious shortcoming as the total times are typically dominated by the BTW stage when using a fast entropy coder. Besides, precompression speeds up entropy coding too.

The text files used in the experiments are described in Table 1. All files were processed as a single block. We have tried a few other files from the Pizza & Chili corpora with similar results (omitted due to lack of space). We use Yuta Mori's `divsufsort` algorithm and implementation (`http://code.google.com/p/libdivsufsort/`) to compute the BWT and the `mtl-sa-8` algorithm in [8] to compute the inverse. The entropy coder is our own experimental coder designed for maximum compression.

The experiments were run on a PC with a 4.2GHz Intel 2600K CPU and 16GiB of 1.6GHz RAM running Linux version 3.0.0-19 (64-bit). The compiler was `g++` (gcc version 4.4.3) executed with the `-O3` option. The execution times are the sum of `user` and `sys` times.

[1] In fact, with less effective entropy coders, the precompression tends to improve compression ratio as it can remove some redundancy that the entropy coder cannot.

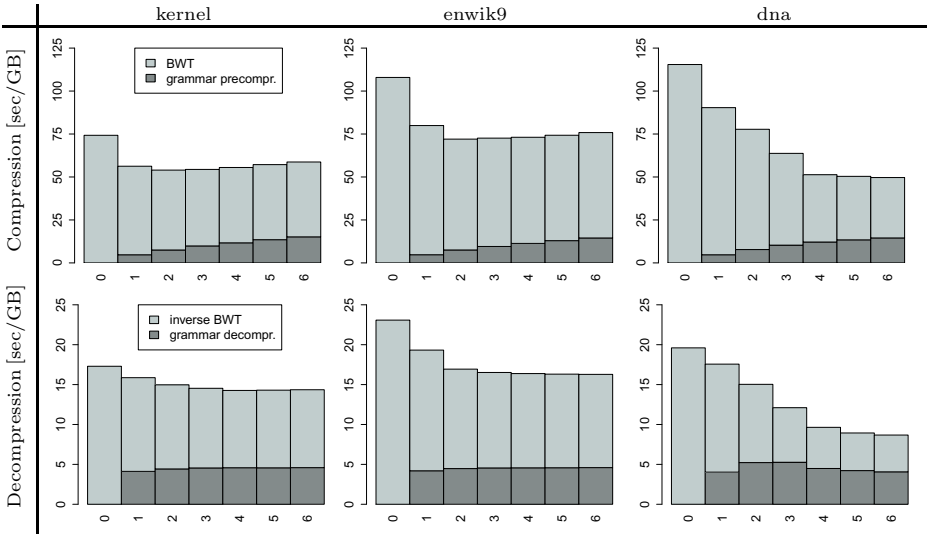

Fig. 2. Times for precompression and BWT stages during compression and decompression. The x-axis labels from 0 to 6 are the number of precompression rounds.

Fig. 2 shows the compression and decompression times. In all cases, at least the first two rounds of precompression improve the combined time. For compression, further rounds can bring a further speed up or a marginal slow down; decompression is never harmed by further rounds. The reduction with four rounds, for example, is more than 17 % in all cases, and more than 50 % for dna.

The compressibility results are shown in Table 2. The effect of the precompression on the compression rate is always less than half a percent of the original file size.

Table 2. Compression rates (bits/char). To demonstrate the effectiveness of the entropy coder, we have included the compression rates for the 7-Zip compressor using two high compression parameter settings (`7z -m0=PPMd:mem=4000m:o32` and `7z -m0=lzma -mx=9 -mfb=273 -md=273 -md=4000m -ms=on`).

	Number of precompression rounds							Other	
Testfile	0	1	2	3	4	5	6	ppmd	lzma
kernel	0.1041	0.1025	0.1018	0.1022	0.1022	0.1022	0.1022	0.0803	0.0643
enwik9	1.3493	1.3617	1.3686	1.3707	1.3712	1.3726	1.3731	1.4288	1.5841
dna	1.7481	1.7444	1.7456	1.7475	1.7498	1.7507	1.7527	1.8418	1.7655

4 Concluding Remarks

In some cases, we have observed improvements in the compression rate too when using the precompressor. First, the precompressor can sometimes remove redundancies that the entropy coder cannot. Second, if the text is processed in smaller

blocks (to speed up the BWT computation or to reduce its memory usage), this can leave redundancies crossing block boundaries undetected. The precompressor with its smaller resource requirements can process the text before the split into blocks and thus remove such redundancies. Furthermore, the precompressor can pack more data into a single block.

The grammar precompressor could be a useful preprocessing stage for other compression methods too. Both the effect of speeding up by reducing the size of the text before executing slower stages of the compression, and improving compression by being able to handle larger portions of the text at a time, are potentially applicable to many compressors.

We have also tried a variant of the Bentley–McIlroy precompression [3]. We did obtain some speed up over no precompression but not as much as with the pair replacement precompressor.

References

1. Abel, J.: Post BWT stages of the Burrows-Wheeler compression algorithm. Softw., Pract. Exper. 40(9), 751–777 (2010)
2. Adjeroh, D., Bell, T., Mukherjee, A.: The Burrows–Wheeler Transform: Data Compression Suffix Arrays, and Pattern Matching. Springer (2008)
3. Bentley, J.L., McIlroy, M.D.: Data compression with long repeated strings. Inf. Sci. 135(1-2), 1–11 (2001)
4. Cannane, A., Williams, H.E.: General-purpose compression for efficient retrieval. JASIST 52(5), 430–437 (2001)
5. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory 51(7), 2554–2576 (2005)
6. Fariña, A., Brisaboa, N.R., Navarro, G., Claude, F., Places, Á.S., Rodríguez, E.: Word-based self-indexes for natural language text. ACM Trans. Inf. Syst. 30(1), 1 (2012)
7. Ferragina, P., Manzini, G.: On compressing the textual web. In: Proc. 3rd Conference on Web Search and Web Data Mining (WSMD), pp. 391–400. ACM (2010)
8. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Slashing the time for BWT inversion. In: Proc. Data Compression Conference, pp. 99–108. IEEE CS (2012)
9. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE 88, 1722–1732 (2000)
10. Mahoney, M.: Large text compression benchmark (July 10, 2012), `http://mattmahoney.net/dc/text.html`
11. Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inf. Syst. 15(2), 124–136 (1997)
12. Skibinski, P., Grabowski, S., Deorowicz, S.: Revisiting dictionary-based compression. Softw., Pract. Exper. 35(15), 1455–1476 (2005)