A Fast Block–sorting Algorithm for lossless Data Compression

DI Michael Schindler

Vienna University of Technology
Karlsplatz 13/1861, A 1040 Wien, Austria, Europe
michael@eiunix.tuwien.ac.at
if .at is transformed to .@ by your sendmail:
michael@hpkloe01.lnf.infn.it
☎ +43 1 3629184 (timezone GMT+1)

Abstract

I describe a fast block–sorting algorithm and its implementation to be used as front–end to simple lossless data compression algorithms like move–to–front coding. I also compare it with widely available data compression algorithms running on the same Hardware. My algorithm achieves speed above comparable algorithms while maintaining the same good compression.
Since it is a derivative from the algorithm published by M. Burrows and D.J. Wheeler the size of the input blocks must be large to achieve good compression. Unlike their method execution speed here does not depend on the blocksize used. I will also present improvements to the backend of block–sorting compression methods.

1 Introduction

Today's popular lossless data compression algorithms are mainly based on the sequential datacompression published by Lempel and Ziv in 1977 [1] and 1978 [2]. There were improvements like in [3] or the development of arithmetic coding [4] but the fundamental algorithms remained the same. Other methods include "Prediction by Partial Matching" (PPM) which was developed in the 1980s. For an overview as in 1990 consult [5]; recent improvements can be found in [6,7](PPMC), [8](PPM∗).

In 1994 M. Burrows and D.J. Wheeler published a new method [10], known as "block sorting", "block reduction" or "Burrows–Wheeler Transform". It is based on a sorting operation that brings together symbols standing in the same or a similar context. Since such symbols do correlate often this correlation can be utilized by simple coding algorithms like a move–to–front coder [11] followed by an entropy coder like huffmann or arithmetic coder. Another possible backend is a locally adaptive entropy coder. P. Fenwick [12] gives a good overview over block–sorting compression.

In this paper the same approach is taken, but compression speed is improved by limiting the size of the context. This results in great compression speed improvements at the cost of a small output file increase and a somewhat slower decompression. The fast compression makes this algorithm specially suitable for on–the–fly compression on file– and wwwservers and for other areas where high throughput is required. It is also well–suited for hardware implementation and is deterministic in time.

In the following sections I will describe the algorithm in more detail, concentrating on the differences to D. Wheelers original algorithm [10]. For a discussion on why and how the resulting output is compressible please see [10], [12] or [13].

In the last section I will present some ideas how to improve the compression of block–sorting output, which are also applicable to the original Burrows–Wheeler Transform.

2 The Burrows–Wheeler Transform

Burrows and Wheeler introduced their transformation by means of rotating the input buffer and sorting the rotations. Finally they output the last character of the sorted rotations. Later in their article they stated that sorting only the suffixes will give the same result.
For ease of understanding I will take a different approach. Try to view the last column (labeled L in [10] and [13]) as a symbol and the first column (labeled F) and all right of it as the context the symbol in column L stands in. Please notice that in this view the context is right of the symbol. What Burrows and Wheeler do is to sort the contexts and output the character that follows (actually: precedes) this context.
For decompression they need an initial context (given implicit through the index of the original row). Starting with this initial context they add the character preceding this context (magically they know where to find it) and get a new context this way. This is repeated until the whole block is reconstructed from the end to the beginning.
So what is the sorting actually used for: It must bring together characters following the same or similar contexts to get good compression and it must ensure the magic to find the right successor to enable decompression. Sorting does both, but there are other methods too. Sorting only limited contexts also brings together symbols in similar context, and that there is a way to undo it efficiently will be shown later.

3 Limited order contexts

Limited order contexts give a special problem to the Burrows–Wheeler transform. Any backtransform requires that the character following each context can be uniquely identified. With limited order contexts the uniqueness of each context is no longer guaranteed, so a method has to be found to distinguish between the same contexts. In the following I will discuss context orders from 0 to n, introducing my algorithm step by step.

3.1 Order 0 contexts

In order 0 contexts all symbols of the input file stand in the same (empty) context. Sorting them by context will not give any result; they might be arranged in any order and the transformed file can not be backtransformed. But there is a surprisingly simple solution to that: if the contexts are equal, the one which comes first in the input buffer sorts lower. Since in this special case all contexts are equal, the symbols are sorted only by their position in the input buffer. The outcome of this transformation is the original file again. So instead of sorting on the context of the context of the context of ... I sort by context and if they are equal I sort by the position in the input buffer (which is unique). In an actual implementation no sorting at all is required if one keeps a separate output buffer for each context. This separation can be done explicit, or space in the output buffer is allocated prior to filling it.

3.2 Order 1 context

Coding using order 1 contexts has no special problems: In a first pass one counts how often each context occurs in the input file. Using this information one allocates sufficient space in the output buffer for all successors of each context. In a second pass these successors are written to the output buffer on the first free position for this context.
Decoding has the problem to find out where the list of successors of each context starts. In the order 0 case this was trivial; since there was only one context successors for this context started at the beginning. Here we need to remember that each symbol of the input occurs in the output exactly once, so the output is a permutation of the input. Since we have order 1 contexts just counting the frequency of each character in the transformed file will give the frequency of each context. Summing up the frequencies of all smaller contexts will give the start of successors for each context. Having the start for each context makes decoding easy: the first unused successor for this context is the correct one.
For decoding an initial context must be given; either explicit or implicit through the index of the start position.

3.3 Order 2 context

Order 2 brings nothing new for the compression part, just the tables to count the frequencies get larger.
In the decompression part there is need to know where each order 2 context starts in the transformed file. There is an easy method for this: Just realize that a context of order i and its successor form a context of order $i+1$. In order 1 we used that an empty context and its successor formed a context of order 1, which was counted then. The same step can be repeated, requiring an additional pass over the input data for each additional character in the context. But there is a better method for higher order contexts presented with the order n context.

3.4 Order n context

Now it is time to do something about the increasing number of context counters: Just omit them. Write a (quick)sort routine that makes a string comparison of n characters, and if all are found equal decide on the position in the block. For low order contexts ($< \log_2(\text{blocksize})$) a different method using radix sort instead is best; see the implementation section for details.
To understand the decompression look at the relation between the Burrows–Wheeler transform and this transform. Both transforms sort contexts lexicographically, and the only difference that may appear is if the first n characters of the context are equal. BWT then sorts on the further context characters, while the proposed transformation sorts on the position in the original file. So the only difference that may appear in the transformation output is a permutation of the characters following the same order n context.
If one applies the inverse BWT to data transformed with the proposed transformation, the inverse BWT algorithm might continue with the wrong successor for this place, but one that will appear at a different place in the original file. So the inverse BWT produces correct results in the sense that each sequence of length n+1 that comes out has a corresponding sequence in the original file. This property is more than what is needed to count order n contexts or backtransform by other methods.
There is only one problem left: The inverse BWT may take a shortcut to the end. This case must be recognized and another startelement for the inverse BWT must be found until all entries of the permutationvector T are used. Since I do not care for the actual start when counting I can start the inverse BWT at any place I want.
Here is the actual algorithm: After locating a startelement (the first unused element of the permutation vector T) process n–1 steps with just filling the context. Then the following is repeated until an used element of vector T is reached: appending the successor to the context, marking the element in the T–vector as used and incrementing the counter for this new context. If it reaches an used element of the T–vector (it is the same where it started incrementing counters) it starts over with a new startelement. Independent of the context size this method requires four passes over the transformed data: One to prepare the T–vector, one to search a start, one to count the contexts and one to produce the output. For lower order contexts there is again a more efficient implementation using radix sort instead of counting all contexts.

3.5 Compression loss with limited order contexts

Experiments showed that for text files the loss when using an order 4 context instead of the unlimited BWT context is in the magnitude of about 0–5%, depending on the postprocessing and the input file. Since the postprocessing is still subject to experiments I can not give better numbers. That the difference is that small might be surprising, but once you consider that the BWT as well as the proposed transform produce blocks built from just a few characters in random sequence and only this random sequence is different, it is not surprising at all. The remaining difference is due to the run–length encoding of zeros after the transformation; BWT is more efficient in collecting zeros together.
Apart from being much faster a limited order sort has additional advantages. Datafiles (like geo in the calgary corpus [15]) usually do not compress well with lossless compressors. Limited order contexts allow to use anything as context; for example the same field of the previous structures in the file or whatever is suitable for the data.

4 Postprocessing

Typically the output of a block–sorting operation is postprocessed by a move–to–front reencoder [11] followed by an entropy coder. There are several ways to improve this step.
First of all it is important to realize that the rank 1 very often appears in pairs after the MTF recoding. This is due to the fact that the sequence aabaa (a being at rank 0, b at rank 1) will look like 00110 after MTF. If one introduces the single change of giving the rank 0 symbol a second chance to prove that it is the most probable one the sequence would look like this: 00100. Practically this is done by maintaining a flag which is cleared by rank 0 and set by all others. Only if the flag is set the a symbol moves to rank 0, otherwise it moves to rank 1. The effect on the output is that the number of rank 0 is increased at the cost of rank 1, giving a skewer distribution. This will pay for large blocksizes only; for small blocksizes the increased cost of moving the right symbol to rank 0 will more than compensate the advantage.
It is also possible to trigger the flushing of the entropy coder statistics synchronously with rank distribution changes. If the input to the MTF recoder changes characteristics (other symbols) it is very likely that the distribution of the symbols will also change and it is time to flush.
At the moment I'm going to experiment with a backend that has the following structure: It will not contain a full MTF step, instead its functionality will be handled by the entropy coder. It will be a three stage coder, the first two stages very similar to [14] but with a modified MTF. The third stage will not perform MTF operations, but will act as full model where the frequencies of symbols presently handled by the other stages will be set to zero. I expect this to have better symbol distributions in the third stage while maintaining the excellent performance of the other stages.

5 Implementation hints

Younger people might not be familiar with the algorithm for sorting punch cards, so it is explained here. Sort them repeatedly with radix sort on one position, starting with the least significant one and ending with the most significant one, making a new pile of all cards after each pass.
What the encoding does is the following: It counts the number of occurrences for each possible order 2 context and calculates starting points for each bucket of the 65536 way radix sort. In this case this needs to be done only once, independent of the number of passes. Then it sorts a pointer to each input character based on the n^{th} and $n–1^{th}$ context character into a new array. Then n is decremented by 2. This is repeated until n is zero. The final array will contain pointers to the input characters sorted by the desired context. Here is why it works: originally the symbols are sorted by the least significant position (the position in the file). So all we need to do is to sort them by all more significant positions. One could use 256 way radix sort, but 65536 way sort needs half the number of passes.

When decompressing a similar method can be used: While building the permutation vector T count the number of occurrences of all possible order 2 contexts. Then calculate starting points like before. While doing the BWT backtransform sort pointers to the decoded characters based on their n and n–1 context into a new array. Then use this new array to build a new permutation vector which will finally give you the backtransformed data.

6 first Results

Following table shows the use of different order models (2, 4 and infinite (BWT)).
It also shows that with large blocksizes (>500kB, the book files) there is some profit using a different MTF coder. For smaller blocksizes the increased cost for moving a symbol to front is greater than the cost saved in keeping it there. I do have large blocks (25–35MB) in the CERN application.
On my home PC I could not give accurate CPU time, but I will run tests with an unix machine at the university. On the PC (including program loading, file reading and writing) ST was found to be 2–30 times faster (depending on the blocksize/filesize) than BWT. For pic without run–length compression the speedup was a factor of about 1000.

The following table contains bits per byte for some files of the calgary corpus [15].

sorting	ST order 2			ST order 4			BWT			bred
ranking	MTF	M1	M2	MTF	M1	M2	MTF	M1	M2	
BIB	2.96	2.91	2.86	2.19	2.16	2.23	2.06	2.08	2.17	2.19
BOOK1	3.32	3.26	3.14	2.59	2.51	2.47	2.47	2.41	2.40	2.98
BOOK2	3.07	3.02	2.98	2.24	2.19	2.22	2.13	2.10	2.14	2.51
GEO	4.70	4.68	4.67	4.80	4.79	4.80	4.78	4.77	4.79	4.89
NEWS	3.41	3.38	3.39	2.69	2.68	2.76	2.60	2.61	2.71	2.94
OBJ1	4.28	4.26	4.30	4.19	4.19	4.25	4.19	4.19	4.25	3.91
OBJ2	2.94	2.93	3.01	2.69	2.72	2.83	2.64	2.69	2.81	2.67
PAPER1	3.17	3.14	3.15	2.61	2.62	2.73	2.56	2.59	2.71	2.58
PAPER2	3.19	3.14	3.09	2.58	2.55	2.61	2.52	2.51	2.59	2.58
PIC	0.98	0.86	0.84	0.96	0.86	0.82	0.80*	0.79*	0.79*	0.82*
PROGC	3.02	2.99	3.06	2.65	2.67	2.80	2.62	2.66	2.80	2.58
PROGL	2.32	2.29	2.34	1.91	1.94	2.05	1.80	1.87	2.00	1.79
PROGP	2.36	2.31	2.40	1.91	1.94	2.08	1.78	1.87	2.04	1.78
TRANS	2.40	2.39	2.44	1.74	1.79	1.94	1.55	1.67	1.84	1.56
means	3.01	2.97	2.98	2.55	2.54	2.61	2.46	2.49	2.58	2.56

ST the proposed transform
MTF a standard move–to–front coder
M1 modified MTF as described in text (good for huge blocks only!)
M2 another variant
* PIC was run–length encoded prior to sorting
All tests (except bred) used a very rapidly adapting arithmetic coder with no run–length compression at all. Using Wheelers method will further improve performance. Details of the final coder will be presented at the conference. I was unable to verify the literature performance given for bred. It might be that bred depends on the machines byte order; I will check with a different computer. My only modification to bred was to set the input and outputfile to binary mode instead the default mode after opening (this is needed for DOS).
The means are given as the unweighted mean of each colum, so that they can be compared to existing literature.

7 Summary

The output produced by this transform has only minor differences to the BWT output, so compression using the same blocksize is about the same. Locality of the input file is better preserved with this new method, while repetition of longer patterns is better preserved with the BWT. Since the execution speed of this method is not affected by blocksize, blocksize is only limited by the available memory. Using larger blocksizes improves compression with block–sorting compression methods, so with a much smaller CPU usage better compression can be achieved using this new transform with large block sizes. Even for small blocksizes this method is several times faster than the BWT, making the postprocessing the limiting step. Another advantage of this algorithm is its insensitivity to repetitions, so a preprocessing of the input is not needed at all. The compression algorithm described is well suited for hardware implementation and is deterministic in time, both properties open a wide area of new applications to block–sorting data compression.
The improvements to the MTF recoder described above and synchronized flushing of the arithmetic encoder statistics, together with run–length encoding described in [14] made some further improvements to block–sorting algorithms for large blocksizes possible. To get small files use the original Burrows–Wheeler transformation, to be fast use this new transformation.

8 Example code

A C–program to demonstrate the proposed transform and backtransform for orders 1 and 2 is available on anonymous ftp at //eiunix.tuwien.ac.at/pub/michael/st/∗.∗ You might want Mark Nelsons programs available on http://web2.airmail.net/markn/articles/bwt/bwtcode.zip (or mirrored at my site) for a complete compression set. On my site there is also example program of order 4 transform, the modified MTF and the combined MTF and ARI (not speedoptimized) if you want to experiment with that. Those are in the invisible directory mentioned in the note to the reviewers (partially not ready for release yet).

9 References

[1] J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory. Vol. IT–23, No. 3, May 1977, pp 337–343.

[2] J. Ziv and A. Lempel: *Compression of individual sequences via variable rate coding.* IEEE Transactions on Information Theory. Vol. IT–24, No. 5, Sept. 1978, pp 530–535.

[3] T.A. Welch: *A technique for high performance data compression.* IEEE Computer, Vol. 17, No 6, June 1984, pp 8–19.

[4] I. Witten, R. Neal and J. Cleary: *Arithmetic coding for data compression.* Communications of the ACM, Vol. 30, 1987, pp 520–540.

[5] T.C. Bell, J.G. Cleary and I.H. Witten: *Text Compression.* Prentice Hall, New Jersey, 1990.

[6] A. Moffat: *Implementing the PPM Data Compression Scheme.* IEEE Trans. Comm., Vol. 38, No. 11, Nov. 1990, pp 1917–1921.

[7] I. Witten, A. Moffat and T.C. Bell: *Managing Gigabytes: Compressing and indexing documents and images.* van Nostrand Reinhold, 1994.

[8] J.G. Cleary, W.J.Teahan, I.H. Witten: *Unbounded Length Contexts for PPM.* Data Compression Conference, DCC–95, Snowbird Utah, March 1995.

[10] M. Burrows and D.J. Wheeler: *A Block–sorting Lossless Data Compression Algorithm.* Digital Systems Research Center, Research Report 124, May 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research–reports/abstracts/src–rr–124.html

[11] J.L. Bentley, D.D. Sleator, R.E. Tarjan and V.K. Wei: *A locally adaptive data compression algorithm.* Communications of the ACM, Vol. 29, No. 4, April 1986, pp 320–330.

[12] Peter Fenwick: *Block Sorting Text Compression – Final Report.* The University of Auckland, Dep. of Computer Science, Technical Report 130, April 1996. ftp://ftp.cs.auckland.nz/out/peter–f/report130.ps

[13] M.R. Nelson: *Data Compression with the Burrows Wheeler Transform.* Dr. Dobbs Journal, Sept. 1996, pp 46–50. http://web2.airmail.net/markn/articles/bwt/bwt.htm

[14] D.J. Wheeler: posted to newsgroup *comp.compression.research*, files available at: ftp://ftp.cl.cam.ac.uk/users/djw3

[15] I.H. Witten und T. Bell: *The Calgary/Canterbury text compression corpus.* Anonymous ftp //ftp.cpsc.ucalgary.ca/pub/text.compression.corpus/text.compression.corpus.tar.Z