Higher Compression from the Burrows-Wheeler Transform by Modified Sorting

Brenton Chapin[*] Stephen R. Tate[†]
Dept. of Computer Science
University of North Texas
P. O. Box 311366
Denton, TX 76203–1366

Abstract

We show that the ordering used in the sorting stage of the Burrows-Wheeler transform, an aspect hitherto ignored, can have a significant impact on the size of the compressed data. We present experimental results showing smaller compressed output achieved with two modifications to the sorting: using a better alphabet ordering and reflecting the sorted strings as in binary reflected Gray coding.

[*]E-mail: `chapin@cs.unt.edu`
[†]E-mail: `srt@cs.unt.edu` — Supported in part by NSF Grant CCR-9409945.

1 Introduction

The Burrows-Wheeler transform (BWT) takes a file of n bytes and creates n permutations of the data by moving the first 1 to n bytes of the data to the end of the remaining data, in effect rotating the data. The n strings of n bytes each are then sorted, which groups similar contexts together, and the last byte of each string is the output data. Since similar contexts are grouped together, this sequence of "last characters" is highly compressible. To compress the data, the output of the transform is run through a Move-To-Front encoder, and the output of that is compressed with arithmetic coding [1].

The choice of Move-To-Front (MTF) coding is important. While contexts are grouped together, there is no per-context statistical information kept, and so the encoder must rapidly adapt from the distribution of one context to the distribution of the next context. The MTF coder has precisely this rapidly adapting quality.

The order of sorting determines which contexts are close to each other in the resulting output, and so the sort order (including the ordering of the source alphabet) can be important in BWT-based compression. We are not aware of any prior investigation into modifying the sorting phase of the Burrows-Wheeler algorithm, and many people seem to consider that a fixed part of the algorithm. For example, in an extensive study of BWT-based compression [3], Fenwick states that "anything other than a standard sort upsets the detailed ordering and prevents recovery of the data" — however, this is not entirely true, as *any* reversible transformation (such as a modified sorting order) can be used for this first phase.

Even with the rapid adaptability of the MTF coder, placing contexts with similar probability distributions close together reduces the cost of switching from one context to another, resulting in reduction in the final compressed size. This dependence on input alphabet encoding is a characteristic that is fairly unique among general-purpose compression schemes. Previous techniques, including statistical techniques (such as the PPM algorithms) and dictionary techniques (represented by LZ77, LZ78, and their descendants), are largely based on pattern matching which is entirely independent of the encoding used for the source alphabet.

It is easy to test a compression algorithm's dependence on alphabet ordering — simply run the source through a randomly chosen alphabet permutation and see how subsequent compression is affected. Using readily available programs `gzip` (version 1.2.4) as a representative of LZ77-based compression and `bzip` (version 0.21) as a representative of BWT-based compression, the following results were obtained in performing this simple experiment. The input file is a 24 bit color version of the popular image of Lena (original size 786,488 bytes).

	Order	
Algorithm	Original	Random
BWT	586,783	671,612
LZ77	730,980	731,170

Note that the random alphabet reordering has very little effect on the dictionary technique, but makes a huge difference to the BWT-based algorithm. Clearly, using an arbitrarily chosen order can have a significant negative impact on the size of the compressed output. In the case of images, like in this test, the natural intensity-level encoding is not arbitrary and is quite natural (and is taken advantage of by image coders such as DPMC), and so it seems unlikely that a modified sorting order would make a significant improvement. However, for text files and other files where the input coding is initially quite arbitrary, it is reasonable to ask whether reordering the sorting stage can produce better compression results. Our experiments show that finding other orders that improve the compression by small amounts is not difficult.

1.1 Previous Work

The paper of Burrows and Wheeler describing the BWT [1] is only barely over 3 years old, and yet the technique has captured the interest of people in both the research community [3] and the popular computer press [6]. There are at least two publicly released programs based on this technique: `bzip`, an implementation by Julian Seward that includes coding improvements suggested by Fenwick [3], and `szip`, an implementation by Michael Schindler that concentrates on speed improvements. Unfortunately, the latter is available only in binary executable format.

Attempted improvements on the original algorithm (BWT followed by MTF coding) have shown very limited success. The most prominent improvements come from the extensive study done by Peter Fenwick [3] in which some improvements were made to the final coding stage of the Burrows-Wheeler algorithm. Our current work, as summarized in this paper, also provides modest improvements in the compression performance, but we focus on the initial sorting phase of the Burrows-Wheeler algorithm. We are not aware of any prior research into this particular aspect.

2 Improving the Sort Order

When the data rotations are ordered in the BWT, the sorting is done in standard lexicographical order[1]. If the initial alphabet encoding is assigned in an arbitrary manner (such as ASCII encoding or opcode encoding in an object file), then the resulting sorted order is also fairly arbitrary. As demonstrated above, the ordering of the characters can make a significant difference in the size of the BWT compressed output. An improvement to the ordering could center on finding a better arbitrary ordering, or perhaps by having a small library of 4 or so orderings and picking the best one based upon some test or user choice. The speed of the algorithm would be preserved and the ordering could be saved in a few bits.

Another way to improve on the order would be to spend time analyzing the data to determine the best ordering. The order would then need to be saved with the data which would take about 211 bytes (actually $\lceil \log_2(256!) \rceil$ bits). The 211 byte overhead may be mitigated somewhat by using that order as an initial ordering for the MTF coder. This idea could in fact be applied to representative data from some large class (such as English text), and the resulting order could be one of the small number of available fixed orderings as described previously. This provides the reordering benefit to commonly encountered classes of data, and yet avoids the overhead of initially selecting the order for each compressed file. Our experiments show that this is indeed useful for classes such as English text.

These ideas are explored in detail in the next two sections.

[1]In this paper, "lexicographical order" always means the standard lexicographical ordering using the *original* alphabet encoding.

2.1 Heuristic Orders

Our first attempt in reordering the input alphabet was simply hand-selecting orderings that seemed to make sense to us heuristically. One of the best heuristic orderings was the one that grouped the vowels together, but kept capital and lower case letters separate as in ASCII. Other seemingly sensible heuristic orderings, such as grouping capitals with their lower case equivalent: "AaBbCc...", did not perform as well. Ordering by frequency of occurrence, which has the advantage of no overhead since the decoder can determine the correct ordering from the data, turned out to be one of the worst orderings.

Much uncompressed data is English or some other human language (text data). A heuristic order optimized for text would be useful. In fact, the alphabetical order of ASCII is not the best order. Experiments show that an order which groups similar symbols near each other gets a small (0.25% to 0.5%) improvement over the ASCII ordering. Symbols that are similar are what one would intuitively expect to be similar. The best heuristic orderings tried on text group vowels, similar consonants, and punctuation together. One hand coded ordering, "AEIOUBCDGFHRLSMNPQJKTWVXYZ" plus a few punctuation groupings ("?!" and "+-,."), does well on text, and, since it is close to the original ordering, it does not usually perform much worse on non-text data than the original order. Table 1 shows the performance of the selected ordering on files from the Calgary Compression Corpus, as well as on the 24-bit version of `lena` and the very large (3,334,517 byte) text file of Les Misérables (`lesm10`), obtained from Project Gutenberg. For comparison, an order computed (with methods discussed in the next section and summarized in Table 3) from text data outside the Calgary Compression Corpus improved compression similarly for the text files of the corpus, but suffered a worse penalty for the non-text files.

2.2 Computed Orders

A good way to analyze the data to determine the best ordering uses the idea that a pair of different byte values that are likely to be followed (or preceded) by the same set of byte values should be close to each other in an ordering of the data. This reduces the "change in probability distribution" overhead encountered in the encoding phase. For example, when "?" and "!" are encountered, it is often at the end of a sentence. Sentences are very often followed by a space or a line feed/carriage return.

To utilize the above idea, consider an ordering of contexts as a walk through the various contexts. The cost of going from one context to the next is related to the dissimilarity of the context probability distributions, and we would like to minimize this cost. For reordering the input alphabet, the problem can be viewed as an instance of the traveling salesperson problem (TSP) in which each vertex is a single character context, and the edge costs are computed based on some probability distribution distance measure. We then try to minimize the cost of the TSP tour. Since this is an NP-hard problem [4], we tried various approximation algorithms in order to select an alphabet reordering, and we also tried various distance measures for computing the edge costs.

For each of the 256 possible characters, create a histogram. Each histogram contains counts of the characters immediately preceding each occurrence in the data of the character represented by that histogram. Characters are "close" to each other if their histograms are "close". In the first distance measure, the "distance" between two histograms is captured by summing the squares of the differences of the logarithms of each of the 256 counts. The second distance measure uses a standard measure from probability theory and information theory, the Kullback-Leibler distance,

File	Original	“aeioubcdgf...”
bib	27,097	26,989
book1	230,247	229,558
book2	155,944	155,515
geo	57,358	57,369
news	118,112	117,734
obj1	10,409	10,402
obj2	76,017	76,062
paper1	16,360	16,221
paper2	24,826	24,705
pic	49,422	49,427
progc	12,379	12,331
progl	15,387	15,304
progp	10,533	10,503
trans	17,561	17,514
total	821,652	819,634
lena	586,783	589,913
lesms10	923,850	920,558

Table 1: Performance of hand-selected heuristic alphabet reordering.

or relative entropy [2]. The third and fourth distance measures are based on distance measures used in the algorithms literature when analyzing the move-to-front algorithm [8] (which is the basis for the coding phase of the BWT-based compressor): the histograms are sorted in decreasing order of frequency, and then the number of “inversions” between the two lists are counted. The third distance measure is precisely the number of inversions, and the fourth distance measure is the logarithm of the number of inversions.

For one approximate solution to the resulting TSP, we used a simple approximation algorithm based on minimum spanning trees due to Rosenkrantz, Stearns, and Lewis [7]. For distance measures satisfying the triangle inequality, the tour produced by this algorithm is guaranteed to be no worse than twice the optimal tour length, but in our case the distances do not necessarily satisfy the triangle inequality and so there is no guarantee on the performance of the algorithm. We also used several of the approximation algorithms included in the `TspSolve` package distributed by Chat Hurwitz [5] — specifically, we used the addition, farinsert, multifrag, and loss techniques from this package. The results of these tests are summarized in Table 2.

Trials indicate that the various TSP algorithms do find orderings that are better (in terms of TSP tour length) than the original alphabet encoding. In this particular text file, the improvement in TSP lengths is roughly reflected in improvements to the compressed output size. Even with the additional overhead of encoding the reordering permutation, the total compressed size is decreased in the better orderings.

For files in the Calgary Compression Corpus, gains were observed for all of the English text files, and substantial gains were obtained for the file `geo`. On the other hand, very little gain was observed for the `pic` file, and reordering resulted in significantly degraded performance for the `obj2`

	Orig order	MST tour	addition	farinsert	multifrag	loss
Orig metric	230,247	229,561	229,921	229,777	231,210	229,458
	230,247	229,351	229,710	229,566	230,998	229,247
	0	210	211	211	212	211
	26,860	9,415	8,824	8,587	10,829	9,300
KL dist	230,247	230,216	230,017	229,868	230,079	230,424
	230,247	230,007	229,801	229,652	229,867	230,212
	0	209	216	216	212	212
	111.71	44.73	39.42	37.59	45.91	37.62
Inv	230,247	229,712	229,455	230,446	229,496	229,780
	230,247	229,500	229,244	230,230	229,281	229,566
	0	212	211	216	215	214
	274,756	147,632	132,702	131,756	133704	131,622
LogInv	230,247	229,712	229,569	229,808	229,496	229,832
	230,247	229,500	229,358	229,597	229,281	229,620
	0	212	211	211	215	212
	682.1	613.5	571.7	572.2	571.8	584.8

Table 2: TSP reordering results using file book1. Numbers in each box, from top to bottom, are the total compressed size, the compressed size of just the reordered data, the size of encoding the reordering permutation, and finally the TSP tour length.

file.

For non-text files, the weight of the order found by TSP was almost always much less than the weight of the lexicographical order, yet the sizes of the compressed data for each ordering did not always correspond. Clearly, the correlation in such cases is weak. Perhaps reordering contexts with more than one character would result in a more direct correlation.

The largest improvement occurs in files that are not text or true color images but have some non-standard organization for which lexicographical order is not very good. Such files would include 256 color images done with a colormap. Perhaps a measure for determining when to apply TSP would be to compare the compressed size for lexicographical order with the compressed size for a random order. If the two are close, then TSP will likely produce a better order.

Table 3 shows the results of using a computed reordering for all the files in the Calgary Compression Corpus. Each file was run through the reordering process using our first distance measure and the farinsert TSP approximation algorithm, and the resulting output size (including the size required to encode the alphabet permutation) is given in the 2nd column. We also performed a test where we took a large amount of English text unrelated to the corpus (obtained from Project Gutenberg) and computed a good general English text ordering from this data. This fixed reordering was then used in encoding all the data files — the results benefit from the fact that you do not have to encode the fixed permutation of the input alphabet with the compressed data. In Table 3, the last column shows the best compression achieved on each file, and this value is marked in bold in each row. The most interesting thing about the ordering computed from the Project Gutenberg files is that it is an excellent selector of files written in English. With one exception, all of the files

	Orig order	TSP (farinsert, 1st metric)	Fixed (text) reorder	Best
bib	27,097	27,199	**26,977**	26,977
book1	230,247	229,777	**229,071**	229,071
book2	155,944	156,077	**155,613**	155,613
geo	57,358	**55,897**	57,565	55,897
news	118,112	118,385	**117,978**	117,978
obj1	**10,409**	10,647	10,511	10,409
obj2	**76,017**	77,450	77,080	76,017
paper1	16,360	16,477	**16,264**	16,264
paper2	24,826	25,132	**24,654**	24,654
pic	**49,422**	49,682	49,518	49,422
progc	**12,379**	12,579	12,427	12,379
progl	15,387	15,571	**15,364**	15,364
progp	**10,533**	10,734	10,568	10,533
trans	**17,561**	17,887	17,663	17,561
Total size	821,652	823,494	821,253	818,139
lena	**586,783**	599,883	600,872	586,783
lesms10	923,850	**920,541**	921,392	920,541

Table 3: Computed alphabet reordering for all files in the Calgary Compression Corpus

from the Calgary Compression Corpus that performed best under this ordering were in fact the English text files (even though they were not used in computing the ordering). The one exception is the LISP program, which when examined was in fact discovered to have large pieces of English text within it in the form of both comments and function names. The large Les Misérables text file also was not best with this fixed text reordering. For such a large file, a data-dependent ordering saves enough space to more than compensate for the overhead of including the ordering.

It is interesting to compare this table with the results for the hand-selected ordering of the previous section. The performance of the fixed, computed ordering is comparable to that of the hand-selected ordering on the text files. This is encouraging for other arbitrarily encoded input sources, suggesting that we do not have to examine and hand-tune orderings for each input source.

2.3 Improving Sorting Using Longer Contexts

Sorting is typically done in lexicographic order using a standard character encoding such as ASCII. For example, the phrases "ayz, aza, azz, baa, baz, bba, bbz, bza, bzz, caa, caz" are sorted in order. But typical sorting is not always the best way to organize the data for BWT, as has already been demonstrated. If the sorting method is changed to put similar strings closer to each other, an improvement of around 0.5% can be achieved, even when done with one of the better orders discussed in the previous section. Binary reflected Gray code minimizes changes in bits between adjacent binary codes. The sorting of strings can be done in an analogous fashion.

Sorting is changed by inverting the sort order for alternating character positions. Let the jth column of the data in the $n \times n$ BWT matrix be the jth character of all the strings created by rotating the data. The first column is sorted as before. But all following columns will be sorted in forward and backward orders according to data from previous columns. Specifically, whenever a character in a column changes between string i and string $i+1$, the sort order of all following columns is inverted. Only the leftmost change (the lowest column) is considered if more than one column changes. Sorting in this fashion will produce an ordered list reflected in a way analogous to the binary reflected Gray codes. Below is an example, using lexicographical order on the phrases given earlier.

		Inversion Point For Columns	
Normal	Reflected	2	3
ayz	ayz		$\surd$
aza	azz		
azz	aza	$\surd$	$\surd$
baa	bza		
baz	bzz		$\surd$
bba	bbz		
bbz	bba		$\surd$
bza	baa		
bzz	baz	$\surd$	$\surd$
caa	caz		
caz	caa		

Notice how the 2nd and 3rd columns are more homogeneous in the reflected ordering. This greater homogeneity also exists in the last column of the BWT, which is the column used as input to the MTF coder. Unlike the simple alphabet reordering, reflection improved the compression of the BWT algorithm on every file tested.

One does not need to extend the reflection of sorting to all n columns to gain improvement in the compression size. As one might expect, experiments show that the greatest improvement occurred when reflection was done on the 2nd column. Reflecting the 2nd and 3rd columns further improved compression and reflecting the 2nd through jth columns where j ranged from 4 up to several hundred usually reaped further improvements, decreasing as j increased until very little change occurred for $j > 8$. This is summarized in Table 4.

The reflected ordering can of course be combined with the alphabet reordering as described in the two preceding sections. Table 5 shows the results of combining reflection (using maximum number of columns) with some of the sort orders from previous sections.

3 Best overall results

Table 6 takes the best compression results from previous tables and shows the methods used on each file to achieve the result. Since much of the corpus is text, the hand coded order "aeiou" is best for most of the files. The TSP schemes did better than ASCII order on text files, but usually not as well as "aeiou". On `geo`, the one non-text file for which lexicographical order was bad, all the TSP schemes found much better orders. For all files (except paper1 when reordered with "aeiou"), reflected-order sorting improved compression.

File	Number of columns sorted in reflected order					
	0	1	2	4	8	max
bib	27,097	27,053	27,032	27,030	27,031	27,051
book1	230,247	230,158	230,130	230,134	230,069	230,074
book2	155,944	155,885	155,809	155,708	155,663	155,666
geo	57,358	56,924	56,850	56,850	56,855	56,859
news	118,112	117,996	117,944	117,899	117,932	117,897
obj1	10,409	10,388	10,384	10,382	10,381	10,381
obj2	76,017	75,978	75,922	75,851	75,817	75,829
paper1	16,360	16,347	16,350	16,347	16,334	16,341
paper2	24,826	24,808	24,819	24,810	24,818	24,828
pic	49,422	49,415	49,405	49,420	49,407	49,406
progc	12,379	12,364	12,357	12,348	12,341	12,340
progl	15,387	15,378	15,357	15,366	15,380	15,355
progp	10,533	10,529	10,536	10,523	10,529	10,527
trans	17,561	17,519	17,525	17,518	17,484	17,488
total	821,652	820,742	820,420	820,186	820,041	820,042
lena	586,783	584,060	582,742	582,657	582,664	582,664
lesms10	923,850	923,608	923,472	923,282	922,975	923,000

Table 4: The result of ordering data based on the reflected ordering described in Section 2.3.

The savings due to the combination of the two main techniques of this paper (alphabet reordering and reflected order sorting) is actually greater than the sum of the savings due to the individual techniques. Considering the individual pieces of the compression algorithm, we estimate that the alphabet reordering is responsible for approximately 67% of the savings, the reflected sorting is responsible for approximately 26% of the savings, and the remaining 7% of savings comes from the combination of the two techniques.

4 Conclusion

Sorting is a central part of compression based on the Burrows-Wheeler transform, and yet standard lexicographic sorting is only one of many possible sorted orderings. In this paper we have considered alternative alphabet orderings based on both hand-selected (heuristic) and structured techniques (such as using a reduction to the traveling salesperson problem to compute an alphabet reordering), as well as reorderings based on larger sequences of characters. Improvements in compressed size were obtained by both alphabet reordering and selective reversal of ordering within columns of the sorted matrix.

Both main techniques add to the compression time, but alphabet reordering adds almost nothing to the decompression time (and reordering with a fixed permutation adds almost nothing to the compression time). While the reflected sorting provided additional improvements, it was slower in both compression and decompression — perhaps better algorithms designed for this modified sorting could improve performance. Even without such improved algorithms, the efficient alphabet

file	aeiou bcdgf	TSP (MST, 1st metric)	TSP (farinsert, 1st metric)	Fixed (text) reorder
bib	26,960	27,406	27,160	26,935
book1	229,391	229,452	229,623	228,938
book2	155,207	155,809	155,957	155,399
geo	56,887	56,081	55,297	57,041
news	117,557	118,109	118,331	117,913
obj1	10,383	10,801	10,639	10,483
obj2	75,818	77,693	77,181	76,922
paper1	16,237	16,503	16,466	16,257
paper2	24,728	25,167	25,166	24,622
pic	49,390	49,011	49,594	49,531
progc	12,300	12,653	12,582	12,419
progl	15,283	15,684	15,555	15,304
progp	10,489	10,805	10,758	10,566
trans	17,463	17,793	17,837	17,631
total	818,093	822,967	822,146	819,961
lena	585,754	598,230	595,720	597,031
lesms10	920,352	919,424	920,067	921,149

Table 5: Combining reflected-order sorting with alphabet reordering.

File	Original BZIP	Our Best Size	method
bib	27,097	26,935	Fixed text + reflect
book1	230,247	228,938	Fixed text + reflect
book2	155,944	155,207	aeiou + reflect
geo	57,358	55,297	TSP(farinsert) + reflect
news	118,112	117,557	aeiou + reflect
obj1	10,409	10,381	reflect
obj2	76,017	75,818	aeiou + reflect
paper1	16,360	16,221	aeiou
paper2	24,826	24,622	Fixed text + reflect
pic	49,422	49,011	TSP(MST) + reflect
progc	12,379	12,300	aeiou + reflect
progl	15,387	15,283	aeiou + reflect
progp	10,533	10,489	aeiou + reflect
trans	17,561	17,463	aeiou + reflect
total	821,652	815,522	
lena	586,783	582,664	reflect
lesms10	923,850	919,424	TSP(MST) + reflect

Table 6: Overall best results

reordering alone is responsible for approximately two-thirds of our compression improvement, and so this technique could have very practical benefits for data that is compressed once and decompressed many times.

References

[1] M. Burrows and D. J. Wheeler. "A Block–sorting Lossless Data Compression Algorithm," SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994.

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

[3] P. Fenwick. "Block Sorting Text Compression — Final Report," The University of Auckland, Department of Computer Science, Technical Report 130, March 1996. Available electronically as `ftp://ftp.cs.auckland.ac.nz/out/peter-f/TechRep130.ps`

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, New York, 1979.

[5] Chat Hurwitz, "Traveling Salesperson Dispersion: Performance and Description of a Heuristic," Cal Poly San Luis Obispo Senior Project, 1992. Software available from the Stony Brook Algorithm Repository at `http://www.cs.sunysb.edu/~algorith/`

[6] M. Nelson. "Data Compression with the Burrows-Wheeler Transform," *Dr. Dobb's Journal*, pp. 46ff, Sept. 1996.

[7] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. "An analysis of serveral heuristics for the traveling salesman problem," *SIAM J. Comput.*, Vol. 6, pp. 563–581, 1997.

[8] D. D. Sleator, and R. E. Tarjan. "Amortized Efficiency of List Update and Paging Rules," *CACM*, Vol. 28, No. 2, pp. 202–208, 1985.