Data Compression with the Burrows-Wheeler Transform

Student CRISTIAN IONIȚĂ
Student MARIUS VETRICI
Student CĂTĂLIN GHIPSE
Faculty of Economic Cybernetics, Statistics and Informatics,
Academy of Economic Studies

Abstract – A very interesting recent development in data compression is the Burrows-Wheeler Transformation. The idea is to permute the input sequence in such a way that characters with a similar context are grouped together. This paper presents an efficient algorithm, suitable for general purpose applications. We show that this way program achieves a better compression rate than other programs that have similar requirements in space and time.

1.Introduction

The most widely used data compression algorithms are based on the sequential data compressors of Lempel and Ziv. Statistical modeling techniques may produce superior compression, but are significantly slower. In this paper, we present a technique that achieves compression within a percent or so of that achieved by statistical modeling techniques, but at speeds comparable to those of algorithms based on Lempel and Ziv's.

Michael Burrows and David Wheeler recently released the details of a transformation function that opens the door to some revolutionary new data compression techniques. The Burrows-Wheeler Transform, or *BWT*, transforms a block of data into a format that is extremely well suited for compression. It does such a good job at this that even the simple demonstration programs I'll present here will outperform state of the art programs.

2.Burrows - Wheeler Transform Basics

The BWT is an algorithm that takes a block of data and rearranges it using a sorting algorithm. The resulting output block contains exactly the same data elements that it started with, differing only in their ordering. The transformation is reversible, meaning the original ordering of the data elements can be restored with no loss of fidelity. The BWT is performed on an entire block of data at once. Most of today's familiar loss-less compression algorithms operate in streaming mode, reading a single byte or a few bytes at a time. But with this new transform, we want to operate on the largest chunks of data possible. Since the BWT operates on data in memory, you may encounter files too big to process in one fell swoop. In these cases, the file must be split up and processed a block at a time.

E	S	E	P	C	C	T

Figure 2.1, A sample data set

For purposes of illustration, we will consider a small data set, shown in Figure 2.1. This string contains seven bytes of data. In order to perform the B-W transform, the first thing we do is treat a string S, of length N, as if it actually contains N different strings, with each character in the original string being the start of a specific string that is N bytes long. (In this case, the word *string* doesn't have the C/C++ semantics of being a null terminated set of characters. A string is just a collection of bytes.) We also treat the buffer as if the last character wraps around back to the first.

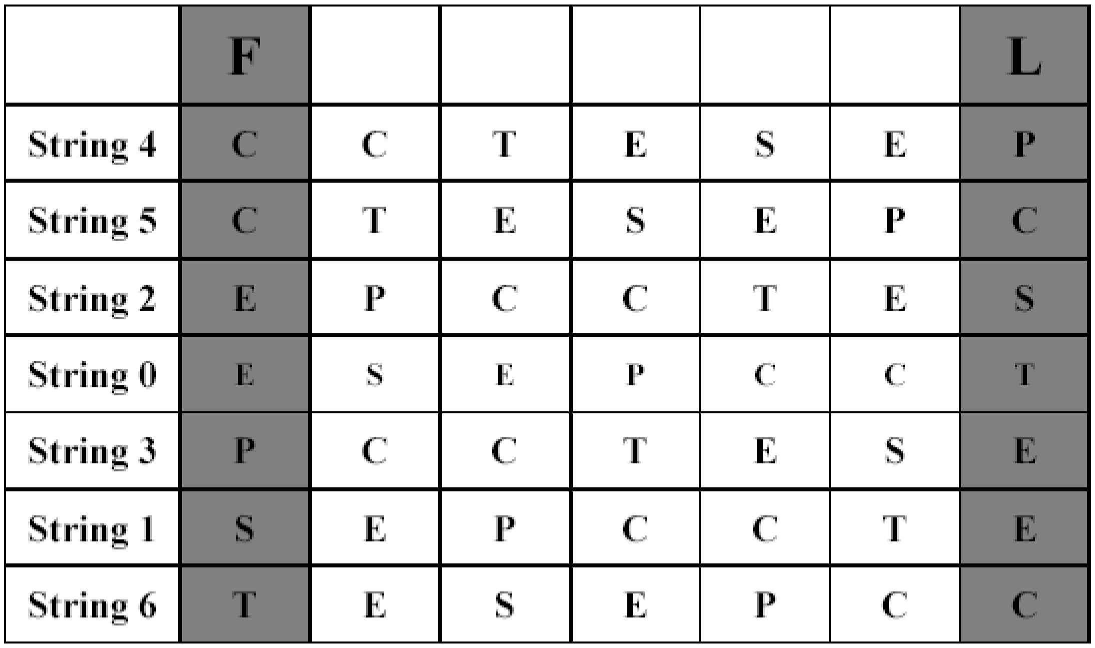
String 0	**E**	**S**	**E**	**P**	**C**	**C**	**T**
String 1	**S**	**E**	**P**	**C**	**C**	**T**	**E**
String 2	**E**	**P**	**C**	**C**	**T**	**E**	**S**
String 3	**P**	**C**	**C**	**T**	**E**	**S**	**E**
String 4	**C**	**C**	**T**	**E**	**S**	**E**	**P**
String 5	**C**	**T**	**E**	**S**	**E**	**P**	**C**
String 6	**T**	**E**	**S**	**E**	**P**	**C**	**C**

Figure 2.2, The set of strings associated with the buffer

It's important to remember at this point that we don't actually make N -1 rotated copies of the input string. In the demonstration program, we just represent each of the strings by a pointer or an index into a memory buffer.

The next step in the B-W transform is to perform a lexicographical sort on the set of input strings. That is, we want to order the strings using a fixed comparison function. In this high level view of the algorithm the comparison function has to be able to wrap around when it reaches the end of the buffer, so a slightly modified comparison function would be needed.

After sorting, the set of strings is arranged as shown in Figure 2.3. There are two important points to note in this picture. First, the strings have been sorted, but we've kept track of which string occupied which position in the original set. So, we know that the String 0, the original unsorted string, has now moved down to row 4 in the array.



	F					L
String 4	C	C	T	E	S	E	P
String 5	C	T	E	S	E	P	C
String 2	E	P	C	C	T	E	S
String 0	E	S	E	P	C	C	T
String 3	P	C	C	T	E	S	E
String 1	S	E	P	C	C	T	E
String 6	T	E	S	E	P	C	C

Figure 2.3, The set of strings after sorting

Second, we've marked the first and last columns' background in the matrix with a gray color and with a special designations F and L. Column F contains all the characters in the original string in sorted order. So our original string "ESEPCCT" represented in F as "CCEEPST".

The characters in column L don't appear to be in any particular order, but in fact they have an interesting property. Each of the characters in L is the *prefix character* to the string that starts in the same row in column F.

The actual output of the B-W transform consists of two things: a copy of column L, and the *primary index*, an integer indicating which row contains the original first character of the buffer B. So performing the BWT on our original string generates the output string L which contains "PCSTEEC", and a primary index of 5.

The integer 5 is found easily enough since the original first character of the buffer will always be found in column L in the row that contains S1. Since S1 is simply S0 rotated left by a single character position, the very first character of the buffer is rotated into the last column of the matrix. Therefore, locating S1 is equivalent to locating the buffer's first character position in L.

3.Two Obvious Questions

At this point in the exposition, there are two obvious questions. First, it doesn't seem possible that this is a reversible transformation. Generally, a *sort()* function doesn't come with an *unsort()* partner that can restore your original ordering. In fact, it isn't likely that you've ever even considered this as something you might like. And second, what possible good does this strange transformation do you?

We will defer the answer to the second question while the reversibility of this transform will be explained. Unsorting column L requires the use of something called the*transformation vector*. The transformation vector is an array that defines the order in which the rotated strings are scattered throughout the rows of the matrix of Figure 3.

The transformation vector, *T*, is an array with one index for each row in column F. For a given row *i*, T[i] is defined as the row where S[i + 1] is found. In Figure 3, row 3 contains S0, the original input string, and row 5 contains S1, the string rotated one character to the left. Thus, T[3] contains the value 5. S2 is found in row 2, so T[5] contains a 2. For this particular matrix, the transformation vector can be calculated to be {1, 6, 4, 5, 0, 2, 3}.

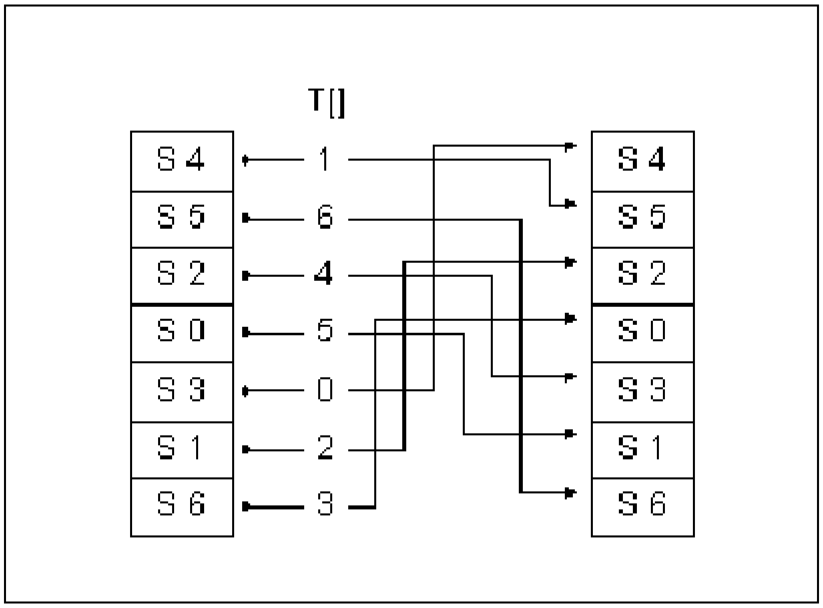


Figure 2.4, The transformation vector routes S[i] to S[i + 1]

Figure 2.4 shows how the transformation vector is used to walk through the various rows. For any row that contains S[i], the vector provides the value of the row where S[i + 1] is found.
The reason the transformation vector is so important is that it provides the key to restoring L to its original order. Given L and the primary index, you can restore the original S0. For this example, the following code does the job:

```
int T[] = { 1, 6, 4, 5, 0, 2, 3 };
char L[] = "OBRSDDB";
int primary_index = 5;
void decode()
{    int index = primary_index;
    for ( int i = 0 ; i < 7 ; i++ ) {
        cout << L[ index ];
        index = T[ index ];}}
```

So now we come to the core premise of the Burrows-Wheeler transform: given a copy of L, you can calculate the transformation vector for the original input matrix. And consequently, given the primary index, you can recreate S0, or the input string.

The key that makes this possible is that calculating the transformation vector requires only that you know the contents of the first and last columns of the matrix. And believe it or not, simply having a copy of L means that you do, in fact, have a copy of F as well.".

	L					
String 4	P	?	?	?	?	?
String 5	C	?	?	?	?	?
String 2	S	?	?	?	?	?
String 0	T	?	?	?	?	?
String 3	E	?	?	?	?	?
String 1	E	?	?	?	?	?
String 6	C	?	?	?	?	?

Figure 2.5, The known state of the matrix given L

Given just the copy of L, we don't know much about the state of the matrix. Figure 2.5 shows L, which I've moved into the first column for purposes of illustration. In this figure, F is going to be in the next column. And fortunately for us, F has an important characteristic: it contains all of the characters from the input string in sorted order. Since L also contains all the same characters, we can determine the contents of F by simply sorting L!

	L	F					
String 4	P	C	?	?	?	?	?
String 5	C	C	?	?	?	?	?
String 2	S	E	?	?	?	?	?
String 0	T	E	?	?	?	?	?
String 3	E	P	?	?	?	?	?
String 1	E	S	?	?	?	?	?
String 6	C	T	?	?	?	?	?

Figure 6, The known state after recovering F

Now we can start working on calculating T. The character 'P' in row 0 clearly moves to row 4 in column F, which means T[4] = 0. But what about row 1? The 'C' could match up with either the 'C' in row 0 or row 1. Which do we select?

Fortunately, the choice here is not ambiguous, although the decision making process may not be intuitive. Remember that by definition, column F is sorted. This means that all the strings beginning with 'C' in column L also appear in sorted order. Why? They all start with the same character, and they are sorted on their second character, by virtue of their second characters appearing in column F.

Since by definition the strings in F must appear in sorted order, it means that all the strings that start with a common character in L appear in the same order in F, although not necessarily in the same rows. Because of this, we know that the 'C' in row 1 of L is going to move up to row 0 in F. The 'C' in row 6 of L moves to row 1 of F.

Once that difficulty is cleared, it's a simple matter to recover the transformation matrix from the two columns. And once that is done, recovering the original input string is short work as well. Simply applying the C++ code shown earlier does the job.

4.The Advantages of Using Burrows-Wheeler Transform

The B-W transform permutes the input sequence in such a way that characters with a similar context are grouped together. This property allows a locally adaptive statistical compression scheme to achieve compression rates that are close to the best known rates. However, the important point is that these rates can be achieved with much less computational effort than previous programs based on statistical modeling techniques. Thus, data compression based on the Burrows-Wheeler Transformation is fast and it leads to good compression results.

5.Conclusions

We have described a compression technique that works by applying a reversible transformation to a block of text to make redundancy in the input more accessible to simple coding schemes. Our algorithm is general-purpose, in that it does well on both text and non-text inputs. The transformation uses sorting to group characters together based on their contexts; this technique makes use of the context on only one side of each character.

To achieve good compression, input blocks of several thousand characters are needed. The effectiveness of the algorithm continues to improve with increasing block size at least up to blocks of several million characters. Our algorithm achieves compression comparable with good statistical modelers, yet is closer in speed to coders based on the algorithms of Lempel and Ziv. Like Lempel and Ziv's algorithms, our algorithm decompresses faster than it compresses.

6.References

[Iv98] Ion Ivan, "Compresia de date", Editura Cison, Bucureşti 1998

[Iv95] Ion Ivan, "Analiza comparativa a algoritmilor de compresie de date" PC World",

nr12., Decembrie 1995

[IE2000] IEEE Transaction on computers, "Universal Data Compression Based on the Burrows – Wheeler Transformation: Theory and Practice" vol. 49, no. 10, October 2000

[Ba98] B. Balkenhol and S. Kurtz, “Universal Data Compression Based on the Burrows and Wheeler Transformation”

http://www.mathematik.uni-bielefeld.de/sfb343/preprints/

[Sa98] K. Sadakane, “A Fast Algorithm for Mking Sufix Arrays and for

Burrows– Wheeler Transformation”, Proc. IEEE Data Compression Conf., pp 129-138, 1998

[Ca99] "Bwt, a transformation algorithm" by Arturo San Emeterio Campos, 1999

http://www.arturocampos.com/ac_bwt.html

[La98] “The Context Trees of Block Sorting Compression”, Proc. IEEE Data Compression Conf., pp. 189-198, 1998

[Yo01] Yokin adim, “Burrows Wheeler Transform FAQ”, 2001

http://arctest.narod.ru/descript/bwt-faq.htm

[Ne00] Mark Nelson, “Data Compression”, 2001

http://www.ddj.com/maillists/compression/do200004cm/do200004cm001.htm

[Ne01] Mark Nelson, “Data Compression with the Burrows-Wheeler Transform”, 2001

http://dogma.net/markn/articles/bwt/bwt.htm

[IJ01] International Journals

http://www.icpads97.korea.ac.kr/html/publications.html

[Sh01] Shin-Cheng Mu, “Burrow – Wheeler Transform and Inverse”

http://web.comlab.ox.ac.uk/oucl/seminars-tt01/extra/mu.html

[Mi99] Peter Bro Miltersen and Sven Skyum, “Data Compression. Lectures.” Fall 1999

http://www.daimi.au.dk/~bromille/DC99/

[Ko99] Joa Koester, “Little essay about the various methods and viewpoints of crunching”, 1999, http://www.woodmann.com/fravia/crunchi8.htm