Burrows-Wheeler's transform

The transform itself does not give any compression, but makes it easier to code the data with a simple coder.

Suppose that we want to code a sequence of length n.

Form all cyclic shifts of this sequence and sort them.

Send the last symbol in each row as a sequence L and the position of the original sequence in the sorted list.

Example

Suppose that we want to code the sequence *bacabba*. We form all cyclic shifts and sort them:

Cyclic shifts	Sorted cyclic shifts
bacabba	*abacabb*
abacabb	*abbabac*
babacab	*acabbab*
bbabaca	*babacab*
abbabac	*bacabba*
cabbaba	*bbabaca*
acabbab	*cabbaba*

The original sequence is found on row 4. The information that is sent is thus the position 4 and the sequence *bcbbaaa*.

Implementation issues

When implementing BWT, it's not necessary to actually form all cyclic shifts. Instead keep the sequence in a circular buffer, form an array of all starting positions and then sort this array.

If we use a general sorting algorithm, we can run into problems when we get long sequences of the same symbol. The comparisons will then take a long time. One solution to this problem is to do runlength coding of the sequence before doing the BWT coding.

Another solution is to exploit the fact that the strings to be sorted are not arbitrary strings, but cyclic shifts. The sorting algorithm can then be tailor made for this situation. See for instance Burrows' and Wheeler's original article for an example of such a sorting algorithm.

Properties

Consider the matrix with sorted sequences, A, and the matrix, A^s we get if we shift all sequences cyclically one step to the right, so that the last symbol ends up in the first position

A	A^s
abacabb	*babacab*
abbabac	*cabbaba*
acabbab	*bacabba*
babacab	*bbabaca*
bacabba	*abacabb*
bbabaca	*abbabac*
cabbaba	*acabbab*

All sequences in A starting with a certain letter are in the same order as they are in A^s. Equivalently, if we consider all sequences in A starting with the same letter, then the suffixes are in the same order as the prefixes of the sequences ending with that letter.

Decoding BWT

This property, which is a result of having sorted cyclic shifts, makes it possible for us to decode the transform, even though we don't have access to all of A.

The sequence we have received, L, is the last column of A. By sorting L we get the first column, F, of A. The letter at position k in F is the one appearing after (circularly) the letter at position k i L.

Decoding BWT, cont.

If we for our example write L and F next to each other we get

L	F
b	a
c	a
b	a
b	b
a	b
a	b
a	c

After b number 1 we have a number 1, after c we have a number 2, after b number 2 we have a number 3, after b number 3 we have b number 1, after a number 1 we have b number 2, after a number 2 we have b number 3 and after a number 3 we have c.

Decoding BWT, cont.

Given the recevied sequence L and the sorted first column F we can thus create a vector T to help us order the symbols in L in the correct order. $T[i]$ points to the index in L where the symbol appearing after $L[i]$ is located. Ie, in the decoded sequence $L[i]$ is followed by $L[T[i]]$ which in turn is followed by $L[T[T[i]]]$, et c.

The coded index I tells us where the sequence begins.

In our example we get $T = [4 \;\; 5 \;\; 6 \;\; 0 \;\; 2 \;\; 3 \;\; 1]$. At position 0 in L we have b number 1 which is followed by a number 1 which can be found at position 4 in L. At position 1 in L we have c which is followed by a number 2 which can be found at position 5 of L, et c.

It's not necessary to actually create F and then sort it. Instead we can just count how many times each letter appears in the sequence and then for each letter in the alphabet, in order, we put the positions in L for that letter into T.

Decoding BWT, cont.

An algorithm for decoding can thus be described in pseudo code as the following, where we have created the vector T from L and F:

$k = T[I]$

$D[0] = L[k]$

for $j = 1$ to $n - 1$
{
    $k = T[k]$
    $D[j] = L[k]$
}

The decoded sequence is now in D.

Alternative decoding BWT

The decoding can also be performed backwards, if we let $T[i]$ point to the position in L for the symbol appearing before $L[i]$. This is equivalent to saying that $T[i]$ is the position in F where $L[i]$ is located.

In our example we get $T = [3 \quad 6 \quad 4 \quad 5 \quad 0 \quad 1 \quad 2]$.

The decoding algorithm becomes:

$k = I$

$D[n-1] = L[k]$

for $j = n - 2$ downto 0
{
    $k = T[k]$
    $D[j] = L[k]$
}

Compression

We still haven't done any compression, rather the opposite, since we besides the sequence also must send a position.

L will be partially sorted, which can be utilized for compression. One way is to use *move-to-front coding (mtf)*.

Start by listing the symbols in some order. For each symbol to be coded we send the index to that symbol, then we place that symbol first in the list. This will (assuming that the list is indexed from 0) give us long runs of zeros, and small values will be more common than large values.

We still haven't done any compression, but the new sequence of indices will have a distribution skewed towards small values, which can be utilized by for instance a Huffman coder or an arithmetic coder. It is also common to do runlength coding of the zeros.

Example

Original text (all whitespace have been replaced by _):

Vi_CARL,_med_Guds_nåde,_Sveriges,_Götes_och_Vendes_Konung_&c._&c._&c.,_arvinge_till_Norge,_hertig_till_Schleswig_Holstein,_Stormarn_och_Ditmarsen,_greve_till_Oldenburg_och_Delmenhorst_&c._&c.,_göre_veterligt:_att,_sedan_Riksens_Ständer_enhälligt_antagit_och_fastställt_den_successionsordning,_varefter_den_högborne_furstes,_Svea_rikes_utkorade_kronprins,_hans_kungl._höghet_prins_JOHAN_BAPTIST_JULII_manliga_bröstarvingar_skola_äga_rätt_till_den_svenska_tronen,_samt_Sveriges_rikes_styrelse_tillträda,_och_denna_grundlag_till_Vårt_nådiga_gillande_blivit_överlämnad,_have_Vi,_i_kraft_av_den_enligt_85_§_i_regeringsformen_Oss_tillkommande_rättighet,_velat_denna_av_Riksens_Ständer_samtyckta_successionsordning_härmed_antaga,_gilla_och_bekräfta,_alldeles_såsom_den_ord_för_ord_härefter_följer:

Example

L (the last letter of the sorted cyclic shifts):

_____snsea.an.sdeiLantgtcccccl8_rtHBC_____O_LTI___RUA_
J__AA__I_____PSJ__:_...tgtNihhd,gsTsllnnvl,ss,,thel,dt,:
atheavrmtlthrnherd,a,a,,,,dgn.,§iesl,stgnsa,tnrtisaae,r,rsnanseeggtseas
,,e5attgdnggnlegtklnrrltt_ssdlmmh__gvmmt_fl___h__g_n&&&&&&uucc
oooooSyaeerräeånanl_____l__nnnånrrudgvdnddgvsrvmmsrrrtbvd
Drsnmddddddvm__ddssvjtdtdvvgtvhgtdtgklkcclhhvr_saäee___
nninriaaiiänörneiiiö__an__niii_cccccc___ggcnn____VV__
wtldrrtlllrrRRtttttggttennvvrrrssgvDllsiislt__eiic_gllllllodlel
Oehnrnlböliiiiiiiaäiäieeollrom_rt_rrloäaaieeeaeereeennmeaa
ääeuroiuiiiuieeddaeeeoiieeaeooaao_______kHskrriiKk__ss
Ntfbhn_eeeeöeaokooooöaä_yguoee__eppeeoäoaotkaouåegaatk__
beennennndeeeese__l_rkkgssn_ånnOeerölra_t___e_tegsgmg
flarietifknnssffesört_____uiS_lsaäämssSSssGrnkbf_aa_aeS_
sSSö_rristt_rr_htltthhrrnnVshhffgrG_

Example

Consider for example the start of the sequences ending with c:

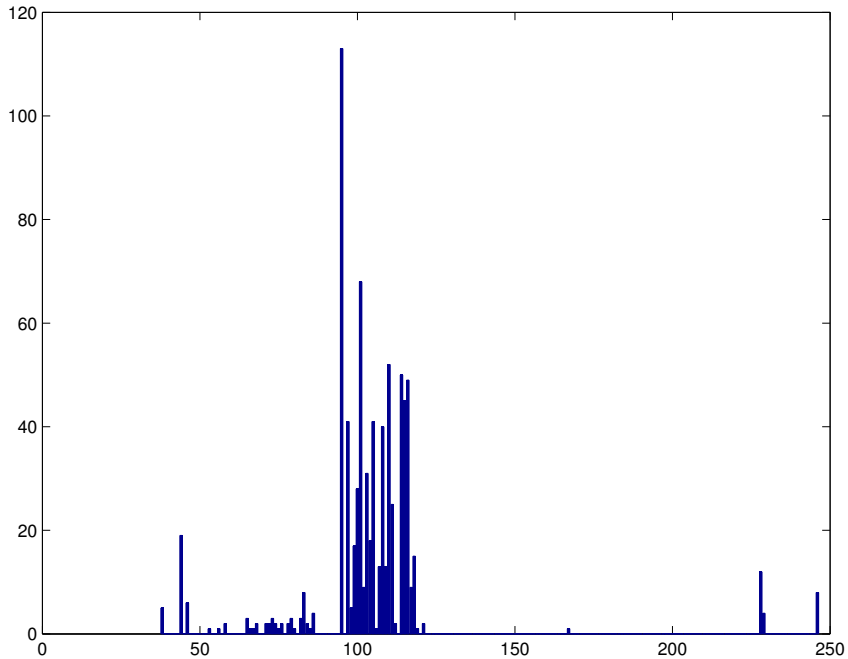
.,_arvinge_till_Norg
.,_göre_veterligt:_a
._&c.,_arvinge_till_
._&c.,_göre_veterlig
._&c._&c.,_arvinge_t
cessionsordning,_var
cessionsordning_härm
essionsordning,_vare
essionsordning_härme
h_Delmenhorst_&c._&c
h_Ditmarsen,_greve_t
h_Vendes_Konung_&c._
h_bekräfta,_alldeles
h_denna_grundlag_til
h_fastställt_den_suc
hleswig_Holstein,_St
kta_successionsordni

Example

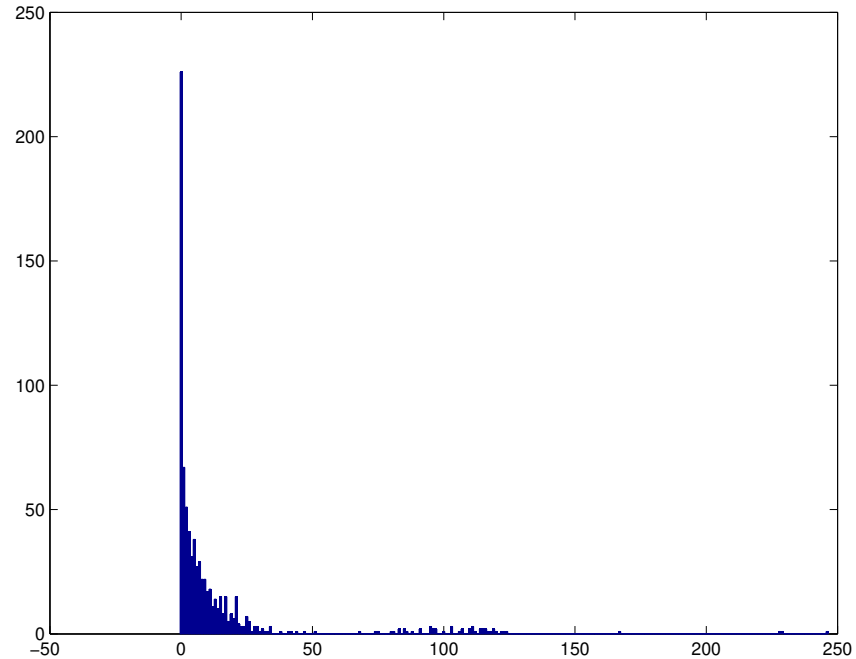
Sequence after mtf-coding:

95 0 0 0 0 115 111 1 103 100 51 1 4 2 4 103 5 107 83 7 7 116 107 1 106
0 0 0 0 111 68 13 116 5 85 80 81 5 0 0 0 0 0 91 1 13 96 88 3 0 0 95 97
86 3 91 1 0 2 0 1 0 5 1 0 0 0 0 0 0 95 97 5 3 0 83 1 27 0 0 16 21 1 96
25 110 0 28 74 6 30 18 1 26 0 28 0 118 2 6 4 0 1 0 11 9 31 5 4 10 5 2
15 31 3 7 7 3 9 29 114 6 10 1 7 4 12 2 8 3 11 11 10 1 1 1 0 0 0 2 14 7
17 4 167 17 9 16 23 5 2 13 9 9 3 11 5 5 4 12 2 9 6 6 0 9 7 6 1 1 4 7 5 1
2 5 0 8 0 8 3 3 5 2 7 0 3 85 4 5 0 6 13 8 2 0 1 15 7 3 5 115 4 5 11 0 2 4
0 22 12 0 9 4 18 0 18 5 0 10 19 4 0 8 4 114 7 2 0 0 7 1 0 7 1 11 75 0 0
0 0 119 0 38 0 117 0 0 0 0 0 29 122 20 20 0 19 0 228 2 229 11 5 1 15 13
0 0 0 0 1 1 0 2 0 0 4 1 7 0 12 20 15 20 2 5 1 0 3 3 21 6 2 21 0 3 3 0 0
21 119 5 7 103 5 6 9 8 5 0 0 0 0 0 110 2 13 0 3 0 5 0 8 120 11 4 1 1 3 0
12 3 2 24 3 3 4 1 2 26 17 1 24 0 2 6 0 7 14 11 11 19 21 21 0 4 0 0 17 0
29 1 7 2 6 0 1 0 6 4 246 5 2 7 5 0 0 4 7 0 7 5 2 0 1 4 0 0 2 12 0 0 0 0 0
1 0 0 14 0 2 4 0 3 0 0 0 19 0 1 0 124 17 16 18 13 0 3 3 0 0 2 0 42 0 3 0
0 0 0 10 0 1 0 14 10 0 17 0 6 0 0 17 0 6 3 22 9 0 4 15 0 1 2 9 13 0 10 5
0 15 3 9 6 0 0 0 0 0 28 14 2 7 1 44 2 21 15 15 1 5 ...

Example



Histogram before mtf



Histogram after mtf

In this exemple we have a sequence of length 790. The longer the sequence, the higher the compression.

Compression of test data

	world192.txt	alice29.txt	xargs.1
original size	2473400	152089	4227
pack	1558720	87788	2821
ppmd	374361	38654	1512
paq6v2	360985	36662	1478
compress	987035	62247	2339
gzip	721413	54191	1756
7z	499353	48553	1860
bzip2	489583	43202	1762

pack is a memoryless static Huffman coder. compress uses LZW. gzip uses deflate. 7z uses LZMA. bzip2 uses BWT + mtf + Huffman coding.

Compression of test data

The same data as the previous slide, but with performance given as bits per symbol. A comparison is also made with some estimated entropies.

	`world192.txt`	`alice29.txt`	`xargs.1`
original size	8	8	8
`pack`	5.04	4.62	5.34
`ppmd`	1.21	2.03	2.86
`paq6v2`	1.17	1.93	2.80
`compress`	3.19	3.27	4.43
`gzip`	2.33	2.85	3.32
`7z`	1.62	2.55	3.52
`bzip2`	1.58	2.27	3.33
$H(X_i)$	5.00	4.57	4.90
$H(X_i \mid X_{i-1})$	3.66	3.42	3.20
$H(X_i \mid X_{i-1}, X_{i-2})$	2.77	2.49	1.55

Tunstall codes

A Tunstall code is a code where all the codewords have the same number of bits, but where the number of source symbols that are coded with each codeword varies between the codewords.

Set a codeword length n (maximum 2^n codewords).

The elements in the code book are strings of symbols from the alphabet.

Start with a code book consisting of the L symbols in the alphabet.

In each step, remove the most probable element in the code book and replace it with the L strings gotten when concatenating the element with each of the L symbols. Calculate the probabilities for the new elements.

Tunstall coding, cont.

In each step we increase the codebook size with $L-1$, so we can do this k times, where k is the largest integer such that

$$L + k(L-1) \leq 2^n$$

Give each element in the codebook a codeword with n bits.
The elements in the codebook are leaves in an L-ary tree, so we can calculate the average depth $\bar{d}$ (symbols/codeword) of this tree. Since each codeword has n bits, the average data rate R is

$$R = \frac{n}{\bar{d}} \quad \text{[bits/symbol]}$$

Tunstall codes are usually a little worse than Huffman codes. They work ok when the alphabet is small, but they are very impractical when you have a large alphabet.