An implementation of block coding.

David Wheeler
Computer Laboratory
Cambridge University
England

October 1995

Introduction

The routines bred (block reduce) and bexp (block expand) will compress a file foo to foo+ and expand the file foo+ to foo. The basis of the method is given in Report SRC 124 of the DEC corporation available by ftp @
(gatekeeper.dec.com: /pub/DEC/SRC/research-reports/SRC-124.ps.Z)

These routines will compress or expand a series of files with some options.

Use

The routines are called by bred/bexp followed by a list of -options and files to be reduced or expanded.

Options

The routines bred and bexp have the following options called by a -x or -xn.
-x0 and -x are equivalent, where x is a letter and n is a decimal number.
-s silent. no displayed information except errors.
-s1 will cause one line of output and is the default.
-s2 give the compression performance on bred.
-s7 will also include the code and length tables from bred
-Kn blocksize in Kilobytes, default 100K.
-Mn blocksize in Megabytes.
-k keep file being compressed or expanded. -k1 restores default
-en (bexp) number of detected errors before no more files tried, default 1
-rn (bexp) limits filter expansion to r times blocksize. Only needed to limit the effect of faulty data. For non faulty files the ratio can be large. The default is 2 which is ok unless a file has a very large proportion of simple repeats.

The options are followed by a list of files to be converted. bred extends the name by + and bexp removes the +.

Options may come between files, but the options for a file must precede it.

Very large files

If a file is too large for the block size then bred breaks it into blocks of about a blocksize long, using about twice blocksize words. bexp requires the blocksize to be as large as that used by bred if the file is that long. It is not made to handle larger blocks than blocksize. Compression improves with size so the largest practical block size should be used.

Limitations

The compression routine will handle no more than 2^{23} -1 characters of one value.

The routine will be very slow for compressing some files.

A simple filter deals with simple repeats, but the routine will be slow if repeats of say ab or abc etc. form a large proportion of a file to be reduced. Reducing the block size will reduce the time needed as the special cases may have time proportional to size^2. The expand time is not affected.

The compression method uses only local redundancy, and does not use the vertical correlations in a pixel display. Thus it only gives a reasonable compression for such data and may not be close to the optimum.

Small files are expanded and there is no test to eliminate this. A zero length file expands to four characters and a single character file expands to seven or eight characters. It is probably not worthwhile testing and eliminating as most file systems will use a somewhat larger block in any case.

There are no options for protection, a simple "w" is used. To change protection a separate chmod command is needed.

The programs are written for a 32 bit computer in C. It is thought to be both little and big endian.

The speed is slow but the compression is close to optimum. For some files the reduction is twice as slow and the expansion four times as slow as a fast compression routine, such as gzip, but the compression is one tenth greater. The current routine has no hand coding to improve the performance. The actual figures will vary according to the compiler, compiler options etc.

Errors

The following may be printed on the standard error stream.
bexp/bred fails, file "file name"
followed by
a) If a file cannot be opened the statement
"file cannot open to read/write " is printed.
b) If the system will not give the space
"not enough space from malloc " is printed.
c) If bexp finds reduced file has no + termination it prints
"no + terminator" is printed.
d) If bexp does not have enough space
"not enough space" is printed.
e) If bexp detects a faulty length table
"false decode tables" is printed.
f) If bexp finds the blocksize too small
"blocksize too small" is printed.
g) If bexp detects an end of file before completion
"data wrong, too long" is printed.
h) If bexp finds the filter expansion exceeds option r times blocksize then
"too many repeats, faulty data else increase option r" is printed.

Changes

Sascha Kratky located and corrected a programming fault. 29-July. Peter Fenwick located the error in the improvement tables, see Appendix.

He also suggests including
#include <stdlib.h>
in the program. This may be necessary on some systems.

Disclaimer

The routines are intended for reference and research use. They made be used for any purpose at the users own risk. They are not guarranteed in any way, but at this time there are no known faults.

End of data needed for use

Comparisons

The routine bred was run against the Calgary reference file set. PPM* is the best of known compression routines.
BW94 is published in the DEC technical report.
bred is the routine described here.

The next gives the values for the program gzip, while the last four columns give the times for compression and expansion for gzip and bred.
The length units are bytes and the columns give bits per byte or seconds. The timing was done on a DEC5000/133.

File	length	PPM*	BW94	bred	gzip	gzip comp secs	gzip exp secs	bred secs	bexp secs
bib	111261	1.91	2.07	2.07	2.52	1.1	.1	2.5	0.6
book1	768771	2.40	2.49	2.60	3.26	11.9	1.3	27.6	4.7
book2	610856	2.02	2.13	2.19	2.71	7.2	0.9	20.1	3.5
geo	102400	4.83	4.45	4.87	5.35	3.1	0.2	3.9	1.7
news	377109	2.42	2.59	2.62	3.07	4.9	.7	11.4	2.3
obj1	21504	4.00	3.98	3.91	3.84	0.2	0.0	0.4	0.2
obj2	246814	2.43	2.64	2.58	2.65	2.8	0.4	6.8	1.8
paper1	53161	2.37	2.55	2.58	2.80	0.5	0.0	1.0	0.2
paper2	82199	2.36	2.51	2.58	2.90	1.0	0.1	1.8	0.4
pic	513216	0.85	0.83	0.83	0.88	3.3	0.5	4.5	1.4
progc	39611	2.40	2.58	2.58	2.68	0.3	0.0	0.6	0.1
progl	71646	1.67	1.80	1.79	1.82	0.6	0.0	1.5	0.2
progp	49379	1.62	1.79	1.78	1.82	0.4	0.0	0.9	0.1
trans	93696	1.45	1.57	1.56	1.62	0.7	0.1	1.9	0.4
means	224402	2.34	2.43	2.46	2.71	27.0	3.1	60.6	12.6

The means given are the means of the columns and have not been weighted by sample size, so that they can be compared directly with existing literature.

The gzip compression and times have also been given. They were made using times from the Unix time and are user+system. They should be taken as no more than a single sample, from times which will vary with compilers, computers, systems etc. See Appendix for some earlier invalid comparisons and their corrections.

Notes on this implementation

The programs have been assembled from various research elements and are not well annotated or tuned.

Description of bred

The data transforms are as follows:
First the data is read and where there are more than two repeats of a character, the sequence is

slightly changed in a recoverable way so that the maximum length of a repeated character is about 31 for a 2^{32} repeats. This avoids a special case and eliminates the slow down of the quicksort routine for repeated characters.

Next the character string is grouped by each character and a set of pointers is generated to each character set. The characters are packed four to a word with a separate word for each character.

The data is sorted by a quicksort variant dealing with the smallest groups first. After each group is sorted the lower three characters are replaced by an integer giving the position in that group. Thus the remaining sorts still work and the depths of comparisons are reduced.

The next or predicted characters are scanned and converted by move to front, and the frequencies established.

From the frequencies we set up a Huffman coding table. The code used is not a Huffman code but one of equivalent performance. We use the lengths of codes generated by a Huffman process, but the codes are derived from the length information and the known ordering of the alphabet. Thus only the lengths are needed to transmit the coding. They are transmitted taking some advantage of the fact that the table is almost monotonic decreasing.

Next the table of lengths is transmitted followed by the encoded data.

If the EOF is reached then the file is complete, otherwise a new sub block is encoded of size up to blocksize.

Next the other files are encoded.

The program also uses routines shell, a simple sort routine, huffgen to generate the coding tables, and QQS, a quicksort type of word sorting routine with pointers.

Nearly all the complication is due to the lack of a sorting routine of good performance and using little space.

Description of bexp

The routine starts by setting up various tables all derived from the length information. The tables len8 and val8 contain 256 entries and give immediate decoding if the code length is 8 or less.

Otherwise three other tables are used to find the length and determine the code value. One test is used for each length and then the decode is determined directly.

Next the rest of the block is read and decoded. The move to front transform is reversed, and the character and that character count is packed into a word. The character may include the EOF=256, so that only 23 bits are available for the count.

The original file is now generated by following pointers constructed from the character, character count in the word, and the start of that character sequence in the sorted file.

The initial filter is reversed restoring long repeat sequences.

The character streeam is scanned to discover if a further sub block exists before a real EOF is met.

The next file is then processed until all are done.

A number of tests are done during the expansion, mainly to avoid the routine going haywire if faulty data is encountered. Faulty data can still give a result which is not detected to be wrong. The result of a single digit error is usually to make the result meaningless, but more errors can sometimes cause an exceedingly long result, which is limited by the ratio option r.

Huffman and other coding

The basic coding used is not Huffman although it is of equal efficiency. The Huffman coding lengths are derived, and then a code constructed from the lengths with the following extra properties.

The maximum length of a code is limited to 23 bits.

For the codes of a given length, the code values are in reverse order of the character set.

The longer codes have the smaller coding values.

The longest code of all zeros is used as the end of file.

These properties are used to simplify the decoding. The program also constructs a table to decode directly all codes of less than nine bits.

As the predicted character list has a very large number of repeats the frequency of zeroes may exceed one half, leading to a lack of efficiency of the code which will need at least one bit for each coding. To avoid this problem, and to use the fact that there is correlation between succesive zeroes, we use a one two grouping of zeroes i.e. immediate repeats.

This works by extending the character set by one value. We increase other codes by 1 and use 0 or 1 to represent groups of zeroes as follows. The first 0/1 gives 1/2 zeroes. The second 0/1 gives 2/4 zeroes. The third 0/1 gives 4/8 zeroes etc. This nearly always gives some correlation gain, but is assymptotically poor. In fact for very large numbers of repeats, a 1,2,3,4 system will give improvements.

Usually, the frequency of the 0's is greater than the 1's, often leading to codes of different length. It is obvious in this case that when the repeat number uses three or more codes, a permutation would save bits. In fact, the coding for 7 to 14 should be 000,001,010,100,011,101,110,111 rather than 000,001,010,011,100,101,110,111. This assumes longer strings are less frequent.

Coding of the length table

The length table is encoded at the beginning of the compressed file in a slightly complicated way, to take advantage of the repeats of the lengths and nulls and the fact that the frequencies seem to obey Zipf's law, so the nth frequency is about k/n, giving a length of about log(n).

The method deals with ascii or small character sets without much loss, as the length table gives lengths for the needed values only. A special bit to indicate ascii seems not worth while.

The lengths are encoded in hex, and the first value is non zero so that between blocks we can have an arbitrary number of zeros.

Groups of nulls are encoded as values from 12-15. 12-15 give 1 to 4 nulls. The next value v>11 gives an additional (v-11)*4 nulls and so on.

Repeats of the same length are dealt with in the same way using values 8-11. Otherwise, we specify the difference between this code and the last, modulo 23. If it is between -4 and 4 we use the values 1-7, excepting the 4. Otherwise we code 4 and follow by a further hex value to give the remaining 16 possibilities.

Repeat filter

Long strings of repeats are removed in a way similar to the 1/2 method used for long strings of zeroes produced by the move to front algorithm.

If a character c is repeated n times making n+1 in all. Then after the first two we encode with c and $c \oplus 1$ giving 1/2,2/4 etc. If the character following the groups of c's is $c \oplus 1$ or $c \oplus 2$, an extra $c \oplus 2$ is inserted.

This maintains most of the predictability, and the losses if any seem small. It is possible for a file to become larger, but this probably enhances the local predictability so that the overall effect is quite small.

Sorting

The character string is sorted by prefix backwards, and a list of pointers initially pointing at every prefix is rearranged so that the list of pointers gives the prefixes in alphabetic order. Thus the text "there are twelve" would be sorted as shown below.

position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
letter	t	h	e	r	e		a	r	e		t	w	e	l	v	e
pointer	10	6	7	3	9	5	16	13	2	14	8	4	1	11	15	12

This assumes that before t there is a start character whose value is lower than the value of space.

First pointers to machine words are set where each word contains four characters backwards arranged. By counting the frequencies of the letters on input we can set pointers in groups to point at those prefixes starting with a, b, c etc.

The groups are arranged in order of size and the smallest group sorted first by the quicksort routine QQS. QQS splits the group being sorted into those that are equal and then uses the next word -four down- to continue the sort. This depth first strategy uses the cache effectively. Many apparently better methods are slow because the cache is not used effectively.

When a group is sorted the three least significant bytes are replaced by the sequence number (plus 32) in that group. Thus any further sort that uses data that group will not need further depth to complete the sort.

In front of the words we place 32 zero words. As no internal prefix can have longer than 32 zeroes, this causes comparisons however deep to end correctly if we also scan the first prefixes and avoid sorting initial zeroes. We do not need to sort them and if we did we would get the wrong type sort! A start of text of -1 rather than a long enough string of zeroes would have avoided this hazard, at the cost of one character beyond 256.

The most significant bit of the packed words is reversed to cause unsigned comparisons using signed long values.

Subroutines

Three subroutines are used. The first is shell, a simple pointer sorting routine. The second is huffgen, which is used to generate the codes and their lengths from a frequency distribution. The third is QQS, a quicksort type of sorting routine using pointers to point at word values.

References

1) Bell, T.C., Cleary, J.G., and Witten, I.H. (1990) *Text Compression.* Prentice Hall, N,J.
2) John G. Cleary, W. J. Teahan, Ian H. Witten *Unbounded length contexts for PPM* pp 52-61. Proceedings DCC'95 Data Compression Conference, Sowbird Utah. Published by IEEE Computer Society Press, Los Alamitos, Calif.

Appendix

Invalid Comparisons

The section below is the early version of the comparisons. Three columns 6/6 8/4 6/2 were invalid due to a program fault, which was detected by Peter Fenwick whose assistance I gratefully acknowledge. Their corrected versions are given at the end.

The routine bred was run against the Calgary reference file set. PPM* is the best of known compression routines.
BW94 is published in the DEC technical report.
bred is the routine described here.
The next three columns were obtained by measurements and make allowance for the size of the coding dictionary. The are derived from bred but use different coding methods to minimise the Huffman coding loss and utilise some of the correlation between adjacent members of the 'next' list.
The six bit was obtained by encoding three pairs of codes as a single unit. The codes were 1/2 (see coding) 1, >1. The >1 required an extra decoding. This further used the context between close values on the 'next string'. The 8/4 was similar but used four bits to predict two pairs. For Huffman coding this was the best for the Calgary set. It balanced the correlation gain against the Huffman coding loss when the predictions became greater than .5.

The next column gives the benefit of arithmetic coding assuming six bits are used for context and one pair is coded. This must be the best apart from the overheads of the coding tables. These were estimated to be the same as for Huffman, but would need to be slightly greater to specify frequencies more accurately.

The next gives the values for the program gzip, while the last four columns give the times for compression and expansion for gzip and bred.
The length units are bytes and the columns give bits per byte or seconds. The timing was done on a DEC5000/133.

File	length	PPM*	BW94	bred	6/6 huff	8/4 huff	6/2 arith	gzip	gzip comp secs	gzip exp secs	bred secs	bexp secs
bib	111261	1.91	2.07	2.07	1.94	1.92	1.90	2.52	1.1	.1	2.5	0.6
book1	768771	2.40	2.49	2.60	2.43	2.39	2.36	3.26	11.9	1.3	27.6	4.7
book2	610856	2.02	2.13	2.19	2.04	2.02	1.98	2.71	7.2	0.9	20.1	3.5
geo	102400	4.83	4.45	4.87	4.78	4.81	4.69	5.35	3.1	0.2	3.9	1.7
news	377109	2.42	2.59	2.62	2.49	2.47	2.50	3.07	4.9	.7	11.4	2.3
obj1	21504	4.00	3.98	3.91	3.81	3.81	3.78	3.84	0.2	0.0	0.4	0.2
obj2	246814	2.43	2.64	2.58	2.44	2.42	2.40	2.65	2.8	0.4	6.8	1.8
paper1	53161	2.37	2.55	2.58	2.43	2.41	2.38	2.80	0.5	0.0	1.0	0.2
paper2	82199	2.36	2.51	2.58	2.42	2.39	2.37	2.90	1.0	0.1	1.8	0.4
pic	513216	0.85	0.83	0.83	0.78	0.77	0.76	0.88	3.3	0.5	4.5	1.4
progc	39611	2.40	2.58	2.58	2.44	2.44	2.41	2.68	0.3	0.0	0.6	0.1
progl	71646	1.67	1.80	1.79	1.64	1.63	1.61	1.82	0.6	0.0	1.5	0.2
progp	49379	1.62	1.79	1.78	1.64	1.63	1.61	1.82	0.4	0.0	0.9	0.1
trans	93696	1.45	1.57	1.56	1.42	1.40	1.39	1.62	0.7	0.1	1.9	0.4
means	224402	2.34	2.43	2.46	2.34	2.32	2.29	2.71	27.0	3.1	60.6	12.6

The means given are the means of the columns and have not been weighted by sample size, so that they can be compared directly with existing literature.

It will be seen that the bred could be improved until it is the same or marginally better than the reference PPM*. However, it would need at least one extra decoding table, and not perform as well on other files, where the extra table needed tends to diminish or even negate the advantage. Thus the 'improvement' has not been implemented.

The gzip compression and times have also been given. They were made using times from the Unix time and are user+system. They should be taken as no more than a single sample, from times which will vary with compilers, computers, systems etc.

Corrected Figures

The table below gives the corrected figures following the wrong figures. There is no allowance for the decoding tables in the results below. It will be seen that to compress as well as ppm* will need more effort. There many known ways of improving the results, each of which will produce some centi bits of gain, such as arithmetic coding, string encoding for numbers of simple repeats as in the DEC report, repeated transmission of coding tables, optimised coding using extra sent information. They tend to make marginal improvements in compression with extra delays in coding and decoding.

file	ppm*	6/6	6/6true	8/4	8/4true	6/2	6/2true
bib	1.91	1.94	2.05	1.92	2.03	1.90	2.02
book1	2.40	2.43	2.54	2.39	2.50	2.36	2.48
book2	2.02	2.04	2.15	2.00	2.12	1.98	2.10
geo	4.83	4.77	4.82	4.81	4.85	4.69	4.73
news	2.42	2.49	2.61	2.47	2.58	2.45	2.56
obj1	4.00	3.81	3.90	3.81	3.91	3.78	3.88
obj2	2.43	2.44	2.56	2.42	2.54	2.40	2.53
paper1	2.37	2.43	2.56	2.41	2.55	2.38	2.52
paper2	2.36	2.42	2.54	2.39	2.52	2.37	2.50
pic	.85	0.78	0.81	0.77	0.80	0.76	0.79
progc	2.40	2.44	2.57	2.43	2.56	2.41	2.54
progl	1.67	1.64	1.76	1.63	1.76	1.61	1.75
progp	1.62	1.64	1.76	1.63	1.75	1.61	1.74
trans	1.45	1.41	1.54	1.40	1.54	1.39	1.52
means	2.34	2.34	2.44	2.32	2.43	2.29	2.40