Huffman Empirical Analysis

Note: The HuffMark program cannot use the myView.update method. For this reason, when testing, the code regarding myView.update was toggled during benchmark.

1. Compressing/uncompressing

bib.txt is used as a sample file for compressing/uncompressing. As shown in myView, bib, the original file size is 890088 bits, while the uncompressed file bib.unhf is also 890088 bits. Both the huff trees and code tables generated during “preprocessCompress” and “uncompress” are also the same. Finally, Diff reported that bib and bib.unhf are the same.

2. Binary files compression vs. text files compression

Both SimpleHuffProcessor and TreeHuffProcessor were used to compress text files in “Calgary” file folder and binary files in “Waterloo” file folder. As can be seen in Table 1 and Table 2, text files (total percent compression ~43%) can be compressed more than binary files (total percent compression ~18%).

Table 1

Text files in Calgary	from (bytes)	SimpleHuffProcessor			TreeHuffProcessor		
		to (bytes)	time	to/from	to (bytes)	time	to/from
bib	111261	73791	0.215	0.66	72880	0.215	0.66
book1	768771	439405	0.89	0.57	438495	0.859	0.57
book2	610856	369331	0.688	0.60	368440	0.684	0.60
geo	102400	73588	0.194	0.72	72917	0.139	0.71
news	377109	247424	0.483	0.66	246536	0.456	0.65
obj1	21504	17081	0.08	0.79	16411	0.033	0.76
obj2	246814	195127	0.366	0.79	194456	0.356	0.79
paper1	53161	34367	0.113	0.65	33475	0.064	0.63
paper2	82199	48645	0.093	0.59	47748	0.089	0.58
paper3	46526	28305	0.104	0.61	27398	0.051	0.59
paper4	13286	8890	0.017	0.67	7977	0.017	0.60
paper5	11954	8461	0.018	0.71	7563	0.018	0.63
paper6	38105	25053	0.057	0.66	24158	0.048	0.63
pic	513216	107582	0.233	0.21	106777	0.225	0.21
progc	39611	26944	0.097	0.68	26048	0.05	0.66
progl	71646	44013	0.107	0.61	43109	0.084	0.60
progp	49379	31244	0.09	0.63	30344	0.059	0.61
trans	93695	66248	0.127	0.71	65361	0.122	0.70
total bytes read		3251493			3251493		
total compressed bytes		1845499			1830093		
total percent compression		43.241			43.715		
compression time		3.972			3.569		

Table 2

binary files in Waterloo	from (bytes)	SimpleHuffProcessor			TreeHuffProcessor		
		to (bytes)	time	to/from	To (bytes)	time	to/from
clegg.tif	2149096	2034591	3.839	0.95	2033920	3.774	0.95
frymire.tif	3706306	2188589	4.37	0.59	2187821	4.225	0.59
lena.tif	786568	766142	1.429	0.97	765471	1.41	0.97
monarch.tif	1179784	1109969	2.076	0.94	1109295	2.026	0.94
peppers.tif	786568	756964	1.443	0.96	756292	1.365	0.96
sail.tif	1179784	1085497	2.072	0.92	1084819	1.982	0.92
serrano.tif	1498414	1127641	2.162	0.75	1126944	2.078	0.75
tulips.tif	1179784	1135857	2.102	0.96	1135182	2.081	0.96
total bytes read		12466304			12466304		
total compressed bytes		10205250			10199744		
total percent compression		18.137			18.181		
compression time		19.493			18.941		

3. Differences between information storages in the header as counts vs. tree
As can be seen in both Table 1 and 2, the tree method compresses files in “Calgary” and “Waterloo” folders a little bit more than the counts method. Yet for most of these big files, the differences are not very dramatic. In these cases, the “header” does not take up as much space comparing with the large “body” of a file. Thus, the differences between counts header and tree header are not very significant.

To further show the differences between the two methods, we used small files such as test.txt (9 bytes), simpleExample.txt (37 bytes) and a.txt (a 238 bytes files only consisted of ‘a’, intentionally designed to compress a lot). In these cases, the header contributes to a large portion of the file, and the tree method compressed much better than the count method. As is shown in Table 3, counts header causes the compressed files to be significantly larger than the original files, with total percent compression ~ -1003%, while the tree method has a total percent compression of ~ 73%.

Table 3

small files	From (bytes)	SimpleHuffProcessor			TreeHuffProcessor		
		To (bytes)	time	to/from	To (bytes)	time	to/from
a.txt	238	1058	0.007	4.45	37	0.042	0.16
simpleExample.txt	37	1041	0.003	28.14	26	0.001	0.70
test.txt	9	1031	0.003	114.56	12	0.001	1.33
total bytes read		284			284		
total compressed bytes		3130			75		
total percent compression		-1002.113			73.592		
compression time		0.013			0.044		

4. Double-compressing

Sometimes, we can gain additional compression by double-compressing an already compressed file. Eventually there is a limit to when compression no longer saves space on ordinary files.

a.txt is a file intentionally designed to compress a lot, because it is a file only consisting of “a”. a.txt can be compressed by TreeHuffProcessor from 238 bytes to 37 bytes, while the compressed file can be double-compressed into 26 bytes. Yet, this resulting file of 26 bytes is no longer worthwhile to recompress, when doing so would save -224 bits.

The worthiness of compressing a file depends on the frequency distribution of each 8-bit element in the file. If there are big differences between the frequencies of these elements, (which is shown by the imbalance of the Huffman Tree), assigning shorter “codes” for the high frequency elements will pay off the potential increase of “code” length of the low frequency elements. In such case, the file is worthwhile to compress. After compression, there will be an altered frequency distribution of each new 8-bit element in the resulting file, and the file may not be worthwhile to recompress.