File Compression Using Huffman coding

The steps to perform compression is as follows:

1. Process Input File (Get_char_count): In this step, the input file is read and the frequency of each character is found by counting the number of appearances of each. The characters are stored in array 'char_list' and the count is stored in 'char_count'. A pseudo-eof character is also decided while reading the input file.

2. Sorting (quicksort): After finding out the frequency, the characters are sorted in decreasing order of frequency. Sorting algorithm used is Quicksort. Sorted list is stored in same arrays.

3. Build forest of trees (create_Forest): Create a tree for each node. Freq of each character is stored in node-> char_count. An array of nodes is used to store the roots of all trees.

4. Create coding tree (build_Coding_Tree): Combine trees of two smallest frequency and repeat until only one tree is left. The resulting tree is coding tree. It saves the root of the tree in *root.

5. Calculate depth of tree (get_depth): Find the depth of coding tree. It is used to build coding table.

6. Build code table (build_Table): Traverse the tree and find out codes of each node. These are stored in a 256*(depth+1) 2-D array. While compressing, codes are copied from this table. Each row of the table represents a character whose ascii value is row number. The code is a series of 1's and 0's followed by -1 signifying end of code. A character which is unused would have -1 in first col of table.

7. Write compressed data to file (write_huff): Read the input file again and convert the code for each character into binary by left shifting the binary code after every col of coding tree is read. Header information and pseudo-eof character is also printed in output file.

The Steps to perform un compression is as follows:

1. Read header (get_num_chars/ get_char_count): Read the header of input file to get the characters, their frequency and the pseudo-eof character used.

2. Re create Coding Tree: Once we have the characters and their count (in sorted order) from the header, we can build the coding tree as done in compression. The same function is reused here.

3. Write Uncompressed Data (unhuff): Once coding tree is generated, keep reading the compressed file and traverse the tree one bit by bit until you reach the leaf. If the leaf does not contain the pseudo-eof character, print it to output file, else stop un compression.

Pseudo EOF character: A pseudo-eof character can be any character which is not present in the input file. I look for characters starting from ASCII code 1 upto 255 until a character which is unused is detected. The first such character encountered is used as pseudo-eof character.

Header: We need to store the tree information in header file of compressed file so that un compression can be done. The contents of header are:

Line 1: <#leaf nodes in tree><pseudo-eof character>

Line 2 onwards: <char i><Freq char i>

As for a medium sized file, the header is less than 0.1% of its size, the compression of header is not done to keep the code simple. Even after compression of header we can barely get 0.01% better compression which is not worth the additional complexity.

Results:

Input File	Size (B)	.huff size(B)	.unhuff size(B)	compression ratio	compression time (sec)	uncompression time (sec)	gzip compression	gzip ratio
text2.txt	155	178	155	-15%	0.00	0.00	142	8%
3183.txt.utf-8.txt	55734	33171	55734	40%	0.01	0.00	22311	60%
39290-0.txt	121093	70357	121093	42%	0.01	0.00	43247	64%
986.txt.utf-8.txt	127913	73702	127913	42%	0.02	0.01	46267	64%
27916.txt.utf-8.txt	130835	75193	130835	43%	0.02	0.01	46936	64%
39295.txt.utf-8.txt	206943	120092	206943	42%	0.03	0.01	79882	61%
600.txt.utf-8.txt	265581	152126	265581	43%	0.03	0.01	99664	62%
39288.txt.utf-8.txt	377195	220283	377195	42%	0.04	0.02	140983	63%
34114.txt.utf-8.txt	443163	260066	443163	41%	0.06	0.02	143577	68%
39293-0.txt	535045	304313	535045	43%	0.06	0.03	201320	62%
39297.txt.utf-8.txt	618615	355713	618615	42%	0.07	0.03	234026	62%
39296.txt.utf-8.txt	627621	359546	627621	43%	0.08	0.03	241322	62%
36034.txt.utf-8.txt	682370	391867	682370	43%	0.08	0.04	254109	63%
39294.txt.utf-8.txt	710134	407366	710134	43%	0.08	0.04	273146	62%
2554.txt.utf-8.txt	1177120	677852	1177120	42%	0.15	0.07	437736	63%
2638.txt.utf-8.txt	1395758	802429	1395758	43%	0.16	0.08	519832	63%
1399.txt.utf-8.txt	2039777	1163726	2039777	43%	0.27	0.12	745666	63%
text3.txt	3150000	2314234	3150000	27%	0.36	0.22	2111249	33%
2600.txt.utf-8.txt	3288707	1871117	3288707	43%	0.38	0.18	1214721	63%
text4.txt	6300000	4627722	6300000	27%	0.72	0.44	4215998	33%
Average Compression Ratio				**38%**				**57%**

Conclusion: The Huffman coding is a simple yet powerful technique to compress files. From the test cases of different types we can see that on average it achieves 38% compression which is pretty good considering the simplicity of algorithm. Although, compared to the advanced commercially used algorithms like gzip, the performance of Huffman coding is only mediocre.

Applications: In Huffman coding, if we change the coding tree slightly, the output changes a lot. Thus apart from compression, Huffman coding can be used for building our own simple encryption techniques where we intentionally modify the coding tree by either inserting some dummy nodes or inter-changing the frequency of certain characters.

Known Problem: There are some memory leaks in huff.c file. A part of malloced memory cannot be free as I get segmentation fault when I try to free that memory and the error message is that the memory has already been freed. I have not debugged the error hence you might get memory leak errors.