Implementing Huffman Coding using Binary Trees in C++

ADITYA DESHPANDE
CID: 01504794

Contents

Introduction to the Project

In ASCII, each character has a fixed length of 8 bits – meaning that an 8 bit binary number corresponds to an integer which is then looked up in an ASCII table in order to find the corresponding character. The first 32 integers (from 0 to 31) are reserved for special control characters, and hence are not relevant to the text itself. Hence, they are omitted from Figure 1.
While ASCII is very effective at being a great standard for text communication, it is also size inefficient. This is because every character, no matter how often or seldom it is used will have a fixed size of 8 bits. For longer strings of text (e.g. large documents) this causes the size of a file to increase, which is not optimal.

To combat this, we use Huffman Encoding. This is an algorithm that reduces the size of each letter based on its frequency in a particular string. As a result, the average size of a character will become much lower than 8 bits, which means that the size of the file will decrease. Huffman encoding is lossless compression.

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
(sp)	32	0040	0x20	@	64	0100	0x40	`	96	0140	0x60
!	33	0041	0x21	A	65	0101	0x41	a	97	0141	0x61
"	34	0042	0x22	B	66	0102	0x42	b	98	0142	0x62
#	35	0043	0x23	C	67	0103	0x43	c	99	0143	0x63
$	36	0044	0x24	D	68	0104	0x44	d	100	0144	0x64
%	37	0045	0x25	E	69	0105	0x45	e	101	0145	0x65
&	38	0046	0x26	F	70	0106	0x46	f	102	0146	0x66
'	39	0047	0x27	G	71	0107	0x47	g	103	0147	0x67
(	40	0050	0x28	H	72	0110	0x48	h	104	0150	0x68
)	41	0051	0x29	I	73	0111	0x49	i	105	0151	0x69
*	42	0052	0x2a	J	74	0112	0x4a	j	106	0152	0x6a
+	43	0053	0x2b	K	75	0113	0x4b	k	107	0153	0x6b
,	44	0054	0x2c	L	76	0114	0x4c	l	108	0154	0x6c
-	45	0055	0x2d	M	77	0115	0x4d	m	109	0155	0x6d
.	46	0056	0x2e	N	78	0116	0x4e	n	110	0156	0x6e
/	47	0057	0x2f	O	79	0117	0x4f	o	111	0157	0x6f
0	48	0060	0x30	P	80	0120	0x50	p	112	0160	0x70
1	49	0061	0x31	Q	81	0121	0x51	q	113	0161	0x71
2	50	0062	0x32	R	82	0122	0x52	r	114	0162	0x72
3	51	0063	0x33	S	83	0123	0x53	s	115	0163	0x73
4	52	0064	0x34	T	84	0124	0x54	t	116	0164	0x74
5	53	0065	0x35	U	85	0125	0x55	u	117	0165	0x75
6	54	0066	0x36	V	86	0126	0x56	v	118	0166	0x76
7	55	0067	0x37	W	87	0127	0x57	w	119	0167	0x77
8	56	0070	0x38	X	88	0130	0x58	x	120	0170	0x78
9	57	0071	0x39	Y	89	0131	0x59	y	121	0171	0x79
:	58	0072	0x3a	Z	90	0132	0x5a	z	122	0172	0x7a
;	59	0073	0x3b	[	91	0133	0x5b	{	123	0173	0x7b
<	60	0074	0x3c	\	92	0134	0x5c	|	124	0174	0x7c
=	61	0075	0x3d	]	93	0135	0x5d	}	125	0175	0x7d
>	62	0076	0x3e	^	94	0136	0x5e	~	126	0176	0x7e
?	63	0077	0x3f	_	95	0137	0x5f				

Figure 1 - ASCII Table

The Process of Huffman Coding

1. Generate a frequency table – detailing each character in the string of text and it's associated frequency.
2. Put the frequency table in descending order based on the frequency, meaning that the character with the highest frequency is at the top and the one with the lowest is at the bottom.
3. Generate a Huffman tree by:
    a. Taking the lowest and second lowest characters and joining them as children of a new parent in a tree.
    b. This parent has no associated character, but its frequency is the sum of the two children.
    c. The parent is placed back into the list in the correct position based on its new frequency.
    d. Repeat this process until the list only has one item in it – this is the parent with the highest frequency and will act as the root node for the tree.
4. Encode the message by finding the first letter of the string in the tree and see the required route to reach it from the root node. For every left branch taken, write a zero and for every right branch taken, write a 1. This is the new code for the letter. The idea is that the most frequently used characters will be at a shallower depth in the tree, meaning that the path to them will be shorter, meaning that their codes will also be shorter.

The process for decoding is simple – just take the path and follow it until you reach a character in the Huffman tree. Then simply move on to the next bit!

As all parent nodes in a Huffman tree have only two child nodes, a binary tree is appears to be a good option to use for a Huffman tree when implementing Huffman coding in C++.

The Encoding Algorithm

Generating a Frequency Table

The first order of business is to generate a table of each of the characters and their corresponding frequencies. A *map* data structure is used for this, in which you use a *key* (the letter) and that maps to a *value* (the frequency). This is handled by the function *frequency*.

FREQUENCY

```
Input: Blank table(map), string.
Output: Full frequency table (map).

for: all characters in ASCII
     for: all characters in the string
          if: current ASCII char == current string char
               count++
     if: count > 0
          insert ASCII char and its count as a pair into the map
```

The penultimate line of the pseudocode is very important. Initially this *if* statement was not there, meaning that a table of all the characters would be generated, even though many of them would just have a frequency of zero. By only allowing non-zero frequencies to be added to the table, the time taken iterating through the table in future functions is lowered (decreasing time complexity).

Generating the Forest

This is handled by the function *generator*. The data structure of a *vector* of pointers to nodes is used for the forest of individual nodes. The forest is essentially a list of nodes – and the function returns a list which

GENERATOR

```
Input: Full frequency table, Empty forest.
Output: Full and ordered forest.

for: all characters in ASCII
     if: the current character is in the frequency table
          declare a pointer to a new node
          set node->character is the current ASCII character
          set left and right pointers to NULL
          set node->frequency to the paired frequency in the table
          place this node in the right ordered place in the forest
```

is ordered by the frequency of the node (ascending order). There is a separate function for ensuring new nodes are placed in the right order within the forest – *order_place*.

ORDER_PLACE

```
Input: The node to be added to the forest (to_push), the existing
forest.
Output: The forest with the new node included in the right place.

if: the forest is empty
      push the node into the zero index of the vector
else:
      for: all the nodes in the forest
            if: to_push->frequency < current->frequency
                  save the index of the current node
                  break out of loop
            if: the loop was broken
                  emplace to_push at the index of the current node
            else: push the node into back of the vector
```

N.B. While the pseudocode uses a *for* loop, the code uses a *while* loop with an iterator, a condition based on the iterator and a condition based on a *bool*. This is done to avoid actually using a *break* statement in the code. The *for* loop in the pseudocode is just to make the logic easier to understand.

Generating the Huffman Tree

Turning the forest into a single binary tree is not handled by a nested function, but instead by a single *while* loop.

FOREST ➔ HUFFMAN TREE

```
Input: Full forest vector.
Output: Forest vector with one index: root node of the Huffman Tree.

while: the tree isn't generated (i.e. more than one node in the forest)
      declare a pointer to a new parent node and set character to NULL
      set parent->frequency to the sum of lowest and second lowest freq.
      parent->left points to lowest, parent->right points to second
      lowest
      remove lowest and second lowest nodes from the forest vector
      place the new parent node into the forest using order_place
```

The single node pointer left in the vector is a pointer to the root node of the Huffman Tree.

Finding Paths and Generating a Huffman Table

It is at this point where the C++ implementation becomes a little more challenging than the real-life version. A human can see all nodes of the tree at once, identify where the letter they want is and then trace the path whereas a computer cannot. A computer must traverse the nodes one by one in order to find the desired character. The computer must also remember the direct path that was taken to reach the node in question.

This challenge is combated by using a recursive post-order traversal algorithm. To best explain the algorithm's general logic, it is best to use the example provided in the template – the Huffman tree of the string *"go go gophers"*. (Figure 2).

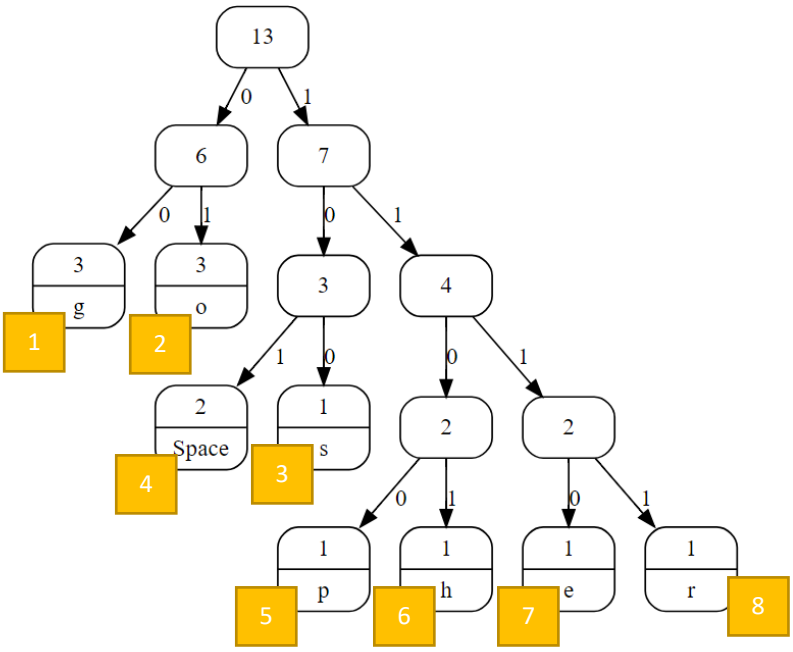


Figure 2 - Huffman Tree

The idea is to visit every leaf node one by one, remembering the path that was taken. Upon the correct node being found, the function will return the correct path. For example, try to find 's' in the tree. The current node pointer starts at the root node and checks if it can move left. If it can then the *find_path* function is called recursively, with the node to the left of the current node as an input parameter. The path is also recorded and passed. It would be foolish to permanently add the direction the node pointer is moving in to the end of the string because the last move may not be correct and would hence need to be undone. Hence, instead of appending to the string permanently, an additional 0 or 1 is concatenated to the end of the path being passed. The algorithm will first visit the node of 'g' as it keeps moving left. Once it realises that is not the correct node, it will check the node to the right of the parent node – 'o'. As the string *result* stores the value of path before concatenation, it serves as a backup.

```
std::string result = path;

if (node->left != NULL)
{
    result = find_path(node->left, c, found, path + "0");
}
if (node->right != NULL && !found)
{
    result = find_path(node->right, c, found, path + "1");
}
if (node->character == c)
{
    found = true;
}
return result;
```

Figure 3 - Code

After exhausting all possibilities to the left of the root node, the top level of the program finally breaks out of the first recursive call within the first *if* statement and then enters the second one. With this, the node pointer moves right the *path* that is passed is 1. After the recursive call the pointer keeps moving left again until it reaches a leaf node. In this case it is the one we are looking for, and the *bool* found is set to true. The numbers on Figure 2 show the order the leaf nodes will be visited in. Once the path is found, it is mapped to the character in a Huffman Table (another *map* structure). This process is repeated for all leaf nodes so codes for all characters in the string are stored.

Generating the Encoded String

From the previous block of code, we have generated a Huffman Table which contains the relevant path for each character. The significance of this table is that it minimises the number of times the tree must be traversed. Without the Huffman Table, the number of times the tree must be traversed using the recursive method detailed on the last page is the number of characters in the string. In contrast, the Huffman Table means that the number of required traversals is equal to the number of unique characters in the string. It is significantly faster to find and store the paths for all the characters, and then look them up in the *map* as opposed to traversing the tree each time.

```
GENERATING ENCODED STRING

Input: Uncompressed message, Huffman Table.
Output: Compressed string of binary digits.

for: all the characters in the message
     define a string path which is originally empty
     look up the char message[i] in the Huffman Table and append the
     paired string to path
```

The Decoding Algorithm

Decoding the message is much simpler than encoding. The string of binary digits that is the encoded message must be used as a guide to move through the Huffman Tree in order to find the correct

```
HUFFDECODE

Input: Encoded Message (enc), Huffman Tree.
Output: Decoded String.
if: the path is not empty
     while: the encoded message is not empty
          set a temporary char current equal to the first char in enc
          delete the first char in enc
          if: current = 0 go left
          if: current = 1 go right
          if: you reach a leaf node (i.e. a character)
          push node->character into the back of the decoded string
          make the node pointer point to the root node again
else:
     for: 0 to node->frequency
          push node->character into the back of decoded.
```

character. Once the algorithm finds a character, it will once again start following the path from the root node(see last *if* statement), meaning that despite each character having different lengths the algorithm can still differentiate between individual characters.

Complexity Analysis

Encoding

Frequency

Viewing the *frequency* function, it is clear that the parameter that would cause the time to increase would be the size of the string. The outer for loop (which runs through all the characters of ASCII) will run for a constant number of cycles as the characters are fixed. However, the number of cycles of the nested for loop is dependent on the size of the string. There is also some variance in runtime based on the diversity of the string (the range of characters within it) however the length is by far a bigger factor. Hence, this function has **linear time complexity: *O(n)***.

Generator

The complexity of the function is dependent on the number of entries in the frequency table – which is dependent on the diversity of the string. If the string has only a few different characters, the if condition is rarely met, decreasing the time taken to run the function. Contrastingly, a worst-case scenario would be if every ASCII character was present in the string, as then the function would have to generate and push a node for every single character (95 in the program). Hence the time increases **linearly** with regard to *n*, the number of unique characters in the string (which is the same as the size of the frequency table). **Linear time complexity: *O(n)***.

Order_Place

It is clear that the runtime of the function is dependent on how many cycles of the loop will take place. For example, the best-case scenario is if the forest is empty, so the node is simply added using *push_back*. However, as the size of the vector increases, the loop will iterate more times. This appears to be a **linear relationship**, suggesting **linear time complexity: *O(n)***.

Generating the Huffman Tree

From looking at the pseudocode, it is obvious that on every iteration of the while loop will decrease the size of the forest by one node (two nodes erased and one emplaced). The exit condition of the loop is when the forest has only one pointer stored within it. Hence there is a **linear relationship** between the number of nodes and the number of loop iterations, suggesting **linear time complexity: *O(n)***.

The Function as a Whole

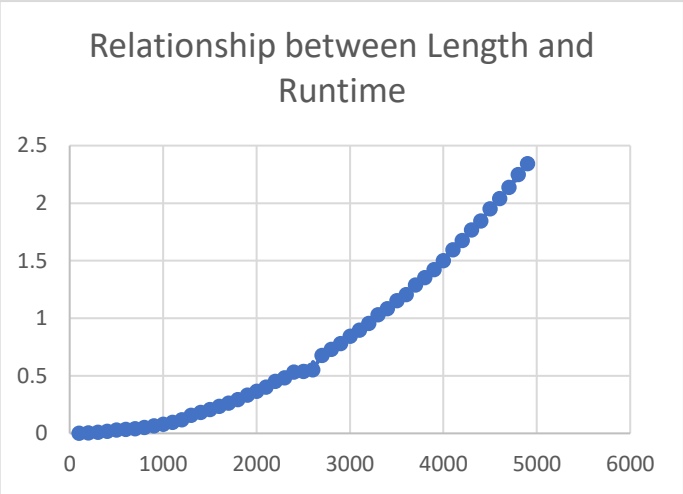
Looking at the functions contained within *huffencode*, along with the recursive algorithm reveals two parameters that could affect the complexity of the Huffman Encoding process**.**

1. **The length of the message string**

   This primarily affects the *frequency* and *generating encoded string* functions, as they contain *for* loops that iterate through the message. For every extra character in the string the loop will have to run an extra time. So theoretically, the relationship between the length of the message string and the runtime should be **linear**.

   However, upon experimental testing the relationship appears to be perfectly quadratic. Results can be found in the table overleaf.

Length	Average Time	Length	Average Time
100	0.00324299	**2600**	0.55165
200	0.00581123	**2700**	0.676375
300	0.0115159	**2800**	0.729189
400	0.019475	**2900**	0.779356
500	0.0286197	**3000**	0.845546
600	0.0353344	**3100**	0.897891
700	0.0419191	**3200**	0.956817
800	0.0533128	**3300**	1.03214
900	0.0663482	**3400**	1.08362
1000	0.0797831	**3500**	1.15449
1100	0.0973498	**3600**	1.20617
1200	0.120085	**3700**	1.29042
1300	0.157835	**3800**	1.35381
1400	0.181357	**3900**	1.42212
1500	0.206625	**4000**	1.49962
1600	0.234342	**4100**	1.59674
1700	0.263605	**4200**	1.6765
1800	0.294744	**4300**	1.76778
1900	0.332369	**4400**	1.84595
2000	0.365907	**4500**	1.95281
2100	0.402531	**4600**	2.0394
2200	0.452649	**4700**	2.1379
2300	0.482986	**4800**	2.25026
2400	0.533231	**4900**	2.34231
2500	0.539527		



The methodology for this test was to create a reference string that was 100 characters long and had a diversity of 25 (all characters from A to Y). This string would then be repeatedly appended to *message*, meaning that the length would increase as: 100, 200, ……,4900. However, as it was the same string repeated the diversity would not change.

The relationship appears perfectly quadratic.

2. **Diversity of the string**

In the worst-case scenario, without a limit on the number of possible characters, the complexity would be *nlog(n)*, because a huge number of nodes would have to be generated. However, in this case, we have a small limit on the diversity, meaning that in theory the length becomes the limiting behaviour once again. Hence in theory this should lead to a **linear complexity *O(n)***. However, after seeing that the relationship between Length and runtime is quadratic, maybe the complexity will be ***O(n).***

Conclusion

From analysing the functions and testing the code, it is evident that the program successfully creates a Huffman tree and successfully supplies an encoded message that can be decoded easily. The experimental example for which the table was given was the worst case scenario for compression rate, as all characters had the same frequency, leading to all the leaf nodes having the same depth. This means that the length of each character's path is the same. Even in this worst case, there was a compression rate of 82%, meaning that a significant reduction in size had been achieved. Hence , we have implemented Huffman Encoding using C++ Binary Trees.