Huffman Coding Project

by
Wenhao Tan, Chenghui Xue, and Chuefeng Vang

Introduction

Nowadays, huge data is constructed everyday. Even though data storage technology is cheaper than few decades ago, it still require costs. Therefore, compression technology was invented. In the project, data structures and algorithms were used to create the Huffman Coding which is a particular type of lossless data compression. The algorithm is interesting because it is a lossless data compression which means it allows the original data to be perfectly reconstructed from the compressed data. Also, during the encoding and decoding, the implementation of the Huffman tree helped apply the methods learned from the CMPE 130 course, Advanced Algorithm. Furthermore, lossless data compression can be used in a lot of fields. For example, music, video, radio or big data. Note that the project only allows .txt file for inputs and outputs, and has no further implementation to the subjects above.

Today, management of big data is changing because the data structures and application relies on the programmers to understand knowledge of advanced data structures similar to that of Huffman Coding. To understand such application, building a huffman tree is useful. Therefore in the approach to build the Huffman Coding, the process used will be encoding and decoding. These approaches will use the algorithmic concept divide and conquer to construct our project. Also, in the program, a selection sort will be used to organize the frequency of characters for easier and better implementation of the Huffman Coding. These approaches helped, due to the fact that instead of building multiple stacks and popping each node out, which would result in a larger use of memory and run time, a sorting algorithm was used on a linked list instead. From there each node would consist of a small tree. The nodes and it's tree would then be used as a stack to build the tree. See methodology and analysis for more details.

Related Work

Huffman's Coding has many related work in correlation to implementation and application of the algorithm. In this implementation an input text was scanned and converted into a linked list which consist of the frequency of character that appeared in the input text. From there the list was sorted by smallest frequency to highest frequency. Next the head node pointer would be placed in the preceding node's children branch, and then removed from the linked list. This would continue until the last node is found. See Methodology for more information. This method can be explained indepthly through the mentioned site as it use an exact implementation as the one explained here: http://michael.dipperstein.com/huffman/ . Other implementation requires the use of stacks for each node, which creates a large use of header, and is more difficult to implement.

Huffman's Coding is also important due to the fact that it is used in compressing large datas. These datas include Zip (big data) files, Jpeg and etc. Using the similar method but with

pixels, Huffman coding can implement an algorithmic solution to compress pixel datas into a blur (larger bytes), and reintroduce the image back to its original form. This is how similar programs can convert large JPEG file size into smaller images like BITMAP. Other implementations include separating color, blur, and brightness into multiple different files. See the link for more information: http://www.shmo.de/mlab/imread.html

Methodology

Our project has three main algorithms: the building of the Huffman tree, saving the information onto a tree, and rebuilding the tree.

In the algorithm for building the Huffman tree, a stack of the characters from an input file will be placed in order from lowest frequency to highest frequency. For every iteration of a while loop, two nodes with lowest frequency will be pop out and combine into a bigger subtree and inserted back into the stack in order. The while loop will end when the stack has only one node left, which means all character nodes have been combined into a Huffman tree. The tree will then be returned as an output of the algorithm.

In order to save the tree for decoding in the future, we use a recursive function to save the information of the tree in a separate file. A “0” will be outputted to the file for any non-leaf node and “1” will be output for all leaf nodes followed by the corresponding character. After the information of current node is outputted, the same recursive function will be called for its left and right child. Starting from the root, all nodes in the tree will be visited and their information will be outputted to the file eventually.

The algorithm of rebuilding the tree is very similar to the previous one, only doing the opposite for every recursive call. If the stream reads a “0”, a new left or right child will be created depending on whether the left child is already created or not. If the stream reads a “1”, more character will be read and the character variable of a new left or right child will be set to that character, again depending on whether the left child is taken already.

The data structure used in these algorithms are a linklist, stack and Huffman tree. Huffman tree is a special form of binary tree, in which parent nodes have combined frequency of their children and characters are only stored in the leaf nodes. The reason and purpose of using the Huffman tree is to reduce the total bits required for the result of encoding. Because of the way we build the tree, nodes with higher frequency are always closer to the root, hence requiring less bits to represent. The reason why we use a stack as a helper tool to build the tree is we can easily pop out the lowest frequency nodes when we need to combine two of them into a bigger subtree.

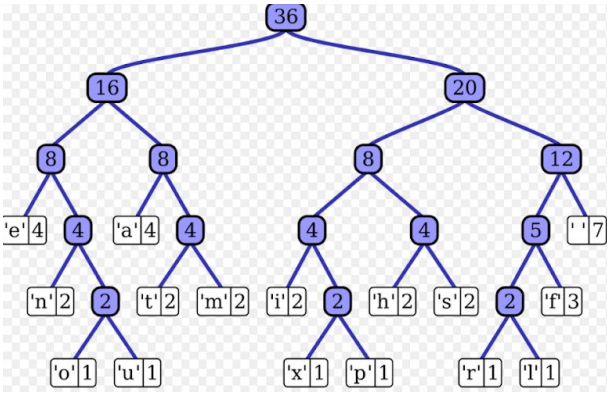

Figure 1: diagram of a Huffman tree

The source code of our project can be found in the repository: https://github.com/xchenghui/Huffman-Coding

Analysis of Algorithms

The two most important algorithms are the building and saving of the Huffman tree. The features they are responsible for are indicated in their names: building the Huffman tree and saving the tree structure in a separate file.

The input of the building algorithm is a stack of the characters from input file in the order from lowest frequency to highest frequency. The output will be a Huffman tree. The pseudo-code is the following:

```
Input: stack a
While (a.size>1) do
    node* left = a.pop()
    node* right = a.pop()
    node* parent = new node
    parent.left=left
    parent.right=right
    a.insert(parent)
Return a
```

The estimated running time of this algorithm is o(m) where m is the number of different characters in the input file, or the size of the stack.

The saving of the tree algorithm uses the current node as the input. The output to the key file will be the output. The pseudo-code is the following:

```
save(node* current){
    if(current.left==null && current.right==null){
        Output 1
        Output current.character
    }
```

```
        Else{
                Output 0
                save(current.left)
                save(current.right)
        }
}
```

The estimated running time of this algorithm depends on the size of the tree, which is at most o(2m) where m is the number of different characters in the input file. Therefore the running time is o(m). The method of rebuilding of the tree depends on how we saved the information. Therefore, the algorithm of rebuilding will be very similar to the saving algorithm and has the same running time.

As we can see, our algorithms depend only on the amount of different characters in the input file. This number is usually considered small compared the total input size of the characters. If we use the ASCII table as an example, the number of different characters should be lower than 127. Therefore, out algorithms don't have a specific need for large memory space.

Results and finding

In order to make the algorithm correct, we do not only test simple string such as AAABBBCCCEEE, but also allowed long poem include space, another line and punctuation. Also, we used ADD1 method to rebuild huffman tree. ADD1 method is that when a pointer travel the tree, if the node is not leaf node, it will write 0 to key (output string), if the node is leaf node, we will write 1 to key and the character to key. Because in huffman coding, characters only can store in leaf nodes. In different field, people used different data compression algorithm. The Lempel-Ziv compression method is the most popular algorithm for lossless storage. We test two cases of Huffman Coding. The first test case is simple string -- AAABBBCCCEEE, and the second test case is a long poem called Daddy by Sylvia Plath.

Here is the first test cases graph:

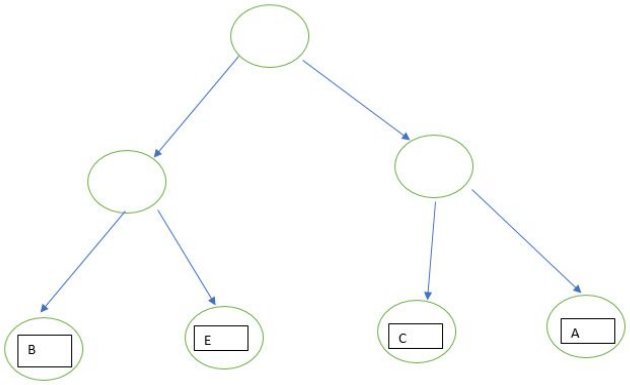

Figure 2. Huffman Tree of AAABBBCCCEEE.

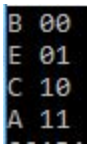

Figure 3. Huffman Table of AAABBBCCCEEE

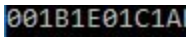

Figure 4. Key (output string to rebuild huffman tree)

Figure 5. Whole first test case.

Here is the second test cases:

```
 00000
S 00001
b 00010
q 000110000
E 000110001
V 000110010
L 000110011
' 0001101
y 000111
t 0010
h 0011
s 01000
I 01001
B 0101000
F 01010010
p 01010011
g 010101
. 0101100
ù 0101101
G 010111000
W 010111001
T 01011101
D 010111100
j 01011110100
C 01011110101
J 0101111011
M 0101111100
- 0101111101
N 01011111100
O 01011111101
z 0101111111
a 0110
d 0111
Y 10000
o 10001
i 100100
P 100101
l 100110
v 100111
w 101000
f 101001
c 101010
k 101011
r 10110
m 101110
, 1011110
A 1011111
u 11000
n 11001
e 1101
  111
```

Figure 6. Huffman table of Daddy

```
000001
1S01b00001q1E01V1l1'1y01t1h0001s1I0001B01F1p1g001.1ù0001G1W1T001D001j1C1J001M1-001N1O1z01a1d00001Y1o001i1P01l1v0001w1f01c1k01r01m01,1A0001u1n1e1
```

Figure 7. Key of Daddy.

Despite the results, some improvement of the project can be improved. During the encoding, we could have significantly reduce the running time of the sorting algorithm by using a quick sort algorithm. This would significantly reduce the sorting to O(Nlogn).

Conclusion

In conclusion, the results show massive improvement such that memory allocation was reduced to nearly minimal levels. The algorithm and run time was also significantly low, despite the use of a selection sort versus a quick sort algorithm. Lastly, the program worked as expected as well. In all the program was a success.