Rutgers, the State University of New Jersey
Math 436 Final Paper
Huffman Coding

Suiliang Ma

Abstract

Huffman coding is an algorithm developed by David A.Huffman while he was a PHD student at MIT. The core idea of Huffman coding is to encode the variables with variable-length code with respect to the probabilities of variables instead of fixed-length code for each of the varaible. While analyzing the advantages of Huffman Coding, this paper also explains the algorithm of Huffman Coding and shows how one can use Huffman Coding to encode and decode the information. At last, this paper discusses the limitations of Huffman Coding.

1 An Introduction of Huffman Coding and Its Influences

Huffman Coding is an algorithm developed by David A.Huffman while he was a PHD student at MIT. In his paper "A Method for the Construction of Minimum-Redundancy Codes," Huffman clearly described the idea of his algorithm, which is what we call Huffman coding today [1]. Huffman Coding is a fundamental and important algorithm in data compression, the subject of which is to reduce the total number of bits demanded to represent information. Huffman Coding even plays an important role in doing the compression in audio and image files [2].

The central idea of Huffman coding is to encode each variable using a variable-length code according to the frequency of each variable. Section 2 explains this idea in detail.

2 Fixed-length Code VS Variable-length Code

2.1 Fixed-length Code

Normally, if one wanted to encode an essay written in English with binary string, he would need to do this following steps:

- Find out how many distinct letters occurred within the essay. Since it is an essay written in English and there are 26 letters in the English alphabet, this man found out **26** distinct letters.
- Find out the number of bits he needed to encode 26 letters. Since $2^4 \leq 26 \leq 2^5$, he needed 5 bits to uniquely represent each of the 26 letters.
- Assign each of the letters with a unique binary string that contains 5 bits.

The entire procedure is called a **fixed-length code** approach as each letter is uniquely assigned with a fixed length of code representing it. Suppose that he found out that the total number of letters contained within the essay was 3000, he would generate an encode string containing 15000 bits overall as he would use 5 bit to encode a single letter. Is that a possible way to reduce the average number of bits needed to encode this essay in order to reduce the total number of bits needed?

Huffman believes that it is possible to do and he proposes his algorithm centered by the idea of variable-length code.

2.2 Variable-length Code

The question that one could come up with respect to fixed-length code approach describd above would be: is that necessary to encode all the letters with the same length of bits?

Huffman believes that it is not necessary to do so. He argues that if the propability of a letter is extremely high, then it could be encoded using fewer bits in order to reduce the total number of bits needed to represent the information. He further articulates that the higher the probability of a letter is, the fewer the bits are needed to encode it [1].

Two questions can be raised based on Huffman's argument. One would be that since the letters will be encoded using different length of bit strings, how could one encode all the information needed and how could one decode the encoded string and receive the correct information. The other question would be that why Huffman's argument reduces the total number of bits needed to represent the information. These two questions will be carefully examined and answered within the coming two sections.

3 The Encoding and Decoding of Information Using Huffman Coding

3.1 Encoding of Information Using Huffman Coding

The encoding of information using Huffman Coding involves two steps. The first step is to build a binary tree called **Huffman Tree**. The second step is to obtain the encoding of information with the help of Huffman Tree. To simplify, the "information" described above means the English alphbet contained within a given document.

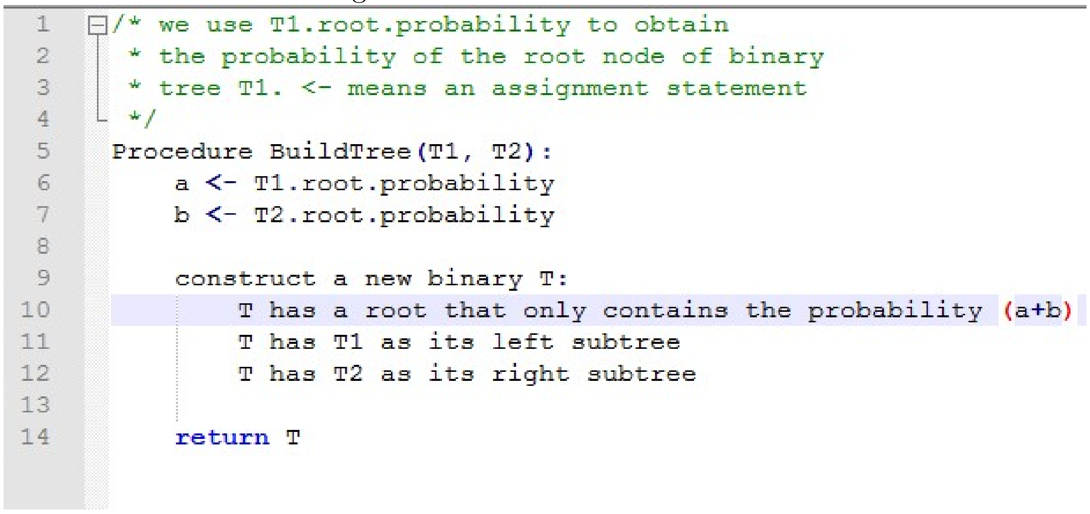
3.1.1 Binary Tree and BuildTree Procedure

We give the recursive definition of a binary tree here:

A binary tree is either empty or consists of a node called root that has a left subtree and right subtree, which are both binary trees with no nodes overlapped [3].

From the definition, we know that a binary tree is uniquely represented by its root. The nodes of a Huffman Tree can be partitioned into **two** categories: one consists of nodes such that every node has a unique English letter and its probability; the other consists of nodes such that every node only has a number representing the probability.

We define a procedure called BuildTree that takes two binary trees $T1$ and $T2$ and builds a new binary tree whose root is the parent of $T1$ and $T2$. The algorithm of BuildTree is described below:



```
/* we use T1.root.probability to obtain
 * the probability of the root node of binary
 * tree T1. <- means an assignment statement
 */
Procedure BuildTree(T1, T2):
    a <- T1.root.probability
    b <- T2.root.probability

    construct a new binary T:
        T has a root that only contains the probability (a+b)
        T has T1 as its left subtree
        T has T2 as its right subtree

    return T
```

We can see that the probability contained in the root of T – the newly built binary tree – is the sum of the probabilities of the root of $T1$ and root of $T2$.

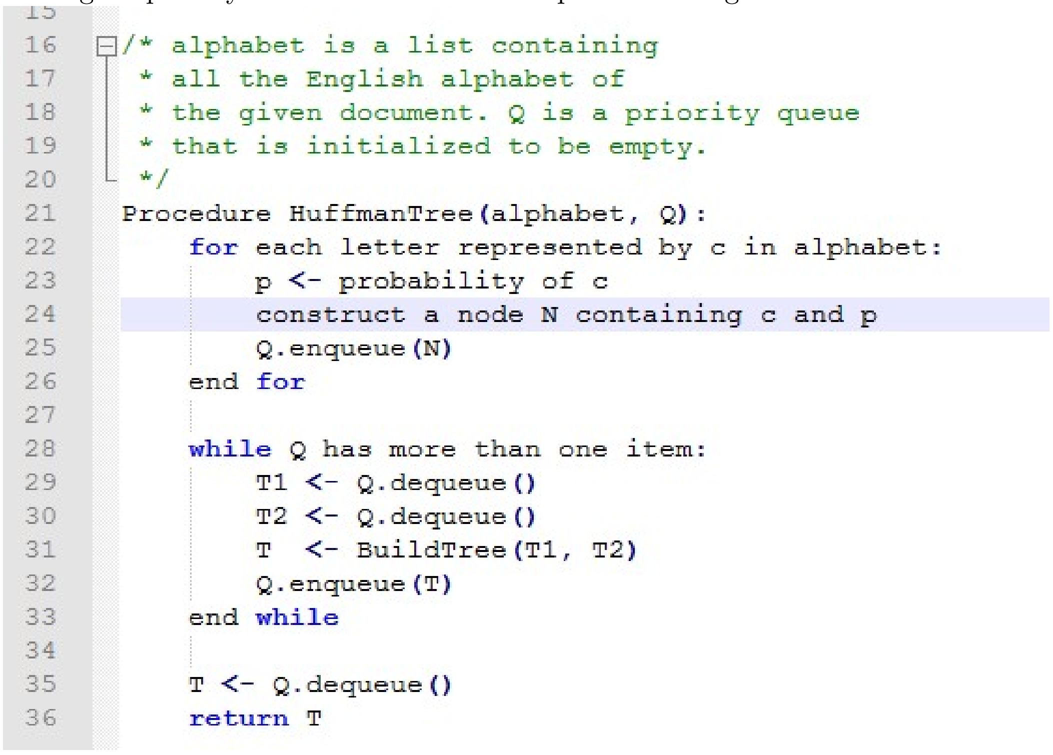
3.1.2 Priority Queue

In order to build a Huffman Tree, a priority queue is needed here. As the name suggests, a priority queue is a queue with priority. All the items contained in a priority queue are arranged with respect to certain priority. A priority supports the following operations:

- enqueue(T): this is an operation such that it inserts item T into the priority queue. Notice that there might be changes within the ordering of all the items after inserting T, but this will be done in this operation.
- dequeue(): this is an operation such that it deletes an item that is placed at the front of the priority queue. After this operation, the total size of the priority queue will decrease by one.

3.1.3 Building Huffman Tree

Now we are ready to decribe the algorithm of building Huffman Tree. We provide a procedure called Huffman Tree that takes a list of English alphabet contained within a given document and an empty priority queue and returns a HuffmanTree that has root T. The priority queue is designed such that the lower the probability of a node is, the higher priority this node will be. The procedure is given below:

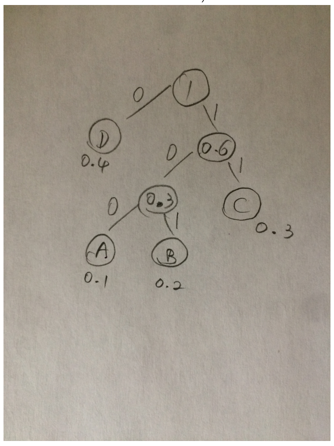


```
/* alphabet is a list containing
 * all the English alphabet of
 * the given document. Q is a priority queue
 * that is initialized to be empty.
 */
Procedure HuffmanTree(alphabet, Q):
    for each letter represented by c in alphabet:
        p <- probability of c
        construct a node N containing c and p
        Q.enqueue(N)
    end for

    while Q has more than one item:
        T1 <- Q.dequeue()
        T2 <- Q.dequeue()
        T  <- BuildTree(T1, T2)
        Q.enqueue(T)
    end while

    T <- Q.dequeue()
    return T
```

3.1.4 Obtain the Encoding String Based on Huffman Tree

Now we can obtain the encoding string with the help of Huffman Tree. With a careful examination of the Huffman Tree, we can see that all the leaf nodes – the nodes that do not have children – contain English alphabet. Therefore, we can determine the encoding string of a letter by searching the node containing such letter. We do this search by starting at the root, which is actually a root with probability 1, appending a 1 to the end of the string if we go to the right subtree, appending a 0 to the end of the string if we go to the left subtree. By doing this, we eventually stop at the node containing the letter we want to encode. Below is an example of such encoding:

suppose that there are four distinct letters within a given document, namely A, B, C and D. We know that the probability of A occurring within the document is 0.1, the probability of B is 0.2, the probability of C is 0.3 while the probability of D is 0.4. According to the HuffmanTree procedure defined in the section 3.1.3, we can obtain the Huffman Tree that looks like this:



As we can see, the encoding strings for A, B, C, D are 100,101,11 and 0, respectively. Then, we can append these encoding strings together to generate the output string that encodes all the letters in the order that they appear. If the document is written as ABBCDCDCDD, then the output string will be 1001011011101101100.

3.2 Decoding of Information Using Huffman Coding

Compared with the encoding part, decoding part is much more direct and straightforward. We will need the Huffman Tree that generates the output string and the output string generated by the exact Huffman Tree as the inputs of this decoding procedure. Let's take the exact Huffman Tree generated in section 3.1.4, and the output string 1001011011101101100 as an example to show the decoding procedure. For the purpose of simplification and precision, we will name the output string 1001011011101101100 as s, the Huffman Tree as T. Then we start the algorithm by traversing s. When seeing an 0, we go one step left in T; we go one step right in T if seeing a 1. We stop and take out the letter if we reach the leaf nodes. Then we will keep doing this until we finish traversing s. In this specific example, we will reach out to the leaf nodes 10 times and generate ABBCDCDCDD, which is the encoded message.

4 Analysis of Huffman Coding

One might ask that why should we choose Huffman Coding than the regular fix-length encoding procedure. The short answer for this question is that Huffman Coding reduces the total number of bits that is required to represent the information. To see this, let's again refer to the example provided in the section 3.1.4.

Now, suppose we use the regular fix-length encoding procedure. Since there are four distinct letters overall, we will need 2 bits to encode these four letters. So, the average number of bits needed for each letter is 2.

Now, assume that we use Huffman Coding to encode the letters. We know that the encoding strings for A, B, C and D are 100,101,11 and 0, respectively. Suppose we have a total number of n letters to encode, then there are $0.1n$ A's, $0.2n$ B's, $0.3n$ C's, $0.4n$ D's. So, the average number of bits needed for each letter is given by

$$\frac{0.1n \times 3 + 0.2n \times 3 + 0.3n \times 2 + 0.4n \times 1}{n}$$

The value of this expression is 1.9, which is less than 2.

Let's consider another example here. Suppose there are four distinct letters to encode, namely A, B,C and D. The probabilities of A, B, C and D that appear within a given document are 0.05, 0.05, 0.1 and 0.8, respectively. We run the entire Huffman Coding algorithm and we will get that the encoding strings for A,B,C and D are 000, 001,01 and 1, respectively. Suppose that this document has a total number of n letters, then we can see that the average number of bits needed for each letter is given by

$$\frac{0.05n \times 3 + 0.05n \times 3 + 0.1n \times 2 + 0.8n \times 1}{n}$$

The value of this expression is 1.3, which is less than 1.9 in the previous example.

To sum up, we can see that though the algorithm will assign more number of bits than that in fixed-length approach to some of the letters, the algorithm assigns less number of bits to the letters that appear much more frequently, in the hope that this will reduce the average number of bits required to represent the information.

Does Huffman Coding always guarantee a reduction of the average number of bits needed to represent the information? Unfortunately, the answer is **no**.

5 Limitations of Huffman Coding

What if all the letters share the same probability? This is a great question.

Suppose that there are m distinct letters and the total number of letters contained in the given document is n. Clearly, the probability of each letter is $\frac{1}{m}$. Since all these m distinct letters will eventually locate at the leaf nodes of the Huffman Tree, and given the situation that they share the same probability, we can see that this Huffman Tree has height $m-1$ as there is exact one leaf node every level except for the last level, where they are two leave nodes. Therefore, we can obtain the average number of bits needed to encode this document by computing this expression:

$$\frac{1}{m} \times ((\sum_{i=1}^{m} i) - 1) = \frac{m^2 + m - 2}{2m}$$

While if we use the fix-length code procedure, the average number of bits needed is

$$\lceil \log_2 m \rceil$$

Let's define functions $f(m)$ and $g(m)$ such that

$$f(m) = \frac{m^2 + m - 2}{2m}$$

$$g(m) = \lceil \log_2 m \rceil$$

We can see that $f(4) = 2.25$, $g(4) = 2$. Moreover, if we want to encode the entire English alphabet, we can see that $f(26) = 13.46$ while $g(26) = 5$.

This means that if m is sufficently large, then we will increase the number of bits needed to encode the information! Therefore, Huffman Coding does not work very well when all the letters sharing the same probability. The situation could be worse if with the increase of the distinct number of letters.

6 Conclusion

We can see that Huffman Coding actually depends on the occurrences of the information. Given an English document as the input, Huffman Coding algorithm would fail to achieve the reduction of average number of bits needed to encode this document if the frequency of each letter appeared in the document is idential.

The examples given in the paper shows that the larger the standard deviation(SD) of the probabilities of the alphabet is, the better that Huffman Coding algorithm will be in terms of reducing the average number of bits needed to encode the document. However, it needs more careful examinations to determine whether this argument is truly held or not.

References

[1] D.A.Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in Proceedings of the IRE, Volume 40, Issue 9, pp 1098-1101, September,1952.

[2] K.Rosen, "Discrete Mathematics and Its Applications," Seventh Edition, McGraw Hill Higher Education, 2012.

[3] S.Venugopal, "Data Structures Outside-In with Java," First Edition, Prentice-Hall Inc, 2006