Huffman Codes

Author 王子腾

Date: 2020-4-30

Chapter 1: Introduction

1.1 Problem Description

For a specified string, the Huffman Codes may be not unique. Although we can build a Huffman Tree according to the character's frequency and acquire an optimized code solution, there are other optimal solutions still. For example :

> {'a'=0, 'x'=10, 'u'=110, 'z'=111} and {'a'=1, 'x'=01, 'u'=001, 'z'=000}
>
> Either of them is an optimal solution for string "aaaxuaxz".

This time, we are given several kinds of "Huffman Codes" produced by students, and our job is to figure out which solutions are correct while others are not.

1.2 Our Tasks

- Receive and transform the INPUT information;
- Find a way to distinguish right Huffman Codes from the wrong ones;
- For every submission, figure out whether it is correct or not.

1.3 Our Thinking

A way to judge if a code method is "Huffman Codes" uses two steps:

> 1. Find the optimal length of the given string and compare it with current solution.
> 2. Test the substring of each coding representation to find if there's ambiguity.

According to the given characters and their frequencies, we can build a Huffman Tree to acquire the optimal length of the given string. After that, the length and substring ambiguity will be the approach to our judgment.

Chapter 2: Algorithm Specification

2.1 Data Structure

To receive the INPUT, we use variables and arrays to store the information.

```
/*Variable definition*/
    int N, M, i, j; //N for number of characters and M for submissions
    int flag = 0;
    int frequency[MaxN]; //Stores the frequency of characters
    char ch;             //The characters themselves are no longer required.
    string str[MaxN];    //An array of strings:used to record each
submission's string
```

To build a Huffman Tree and get optimal length, we define two structures, of which one is to represent a node, the other is to build a heap.

Node Structure

```
typedef struct HNode *HTree;
struct HNode
{
    int weight;        //The weight of each node
    HTree left, right; //Left child and Right child
};
```

Heap Structure

```
typedef struct Heap *MinHeap;
struct Heap
{
    HNode data[MaxN]; //The array begins with the index of 1,the first element is used as sentinel
    int size;
};
```

2.2 Algorithm

2.2.1 Overall

Having saved the INPUT, we can build a Min-Heap to sort the characters according to their frequencies first.

Then, we can create a Huffman Tree and get the optimal length for target string, which will be used to test the correctness later.

For each submission, we calculate it's length representing the whole string :

$$len_i = frequency_i \times size(str_i)$$

$$len = \int_1^N len_i di$$

Compare it with the optimal length and check if there's ambiguity.

Notion: To be more clear, the pointer operation " -> " is substituted by " . " in pseudo-code.

2.2.2 Min-Heap

This part is going to build a Min-Heap according to the character's frequency.

Pseudo-code

Algorithm 1: Overall

```
for i<-0 to N-1
    INPUT: ch frequency
    create new node temp
    Initialise temp's pointer NIL
    temp.weight <- frequency
    Insert temp into Heap
```

Algorithm 2 : Insert

> Parameter : H temp

```
H.size, i <- H.size+1
while temp.weight < H.data[i/2].weight
    H.data[i/2] move down
H.data[i] <- *temp
```

2.2.3 Huffman Tree

This part is to create a Huffman Tree and get the optimal length.

We use Heap to store the minimal

Pseudo-code

Algorithm 1: Overall

```
create new node T <- CreateHuffmanTree(H)
record length <- Wpl(T,0)
```

Algorithm 2 : Create Huffman-Tree

> Parameter : H

```
size <- H.size
for i<-0 to size-1
    create new node T
    initialise T's pointer by DeleteMin(H) twice
    initialise T's weight
    Insert T back to H
create new node T <- DeleteMin(H)
```

Algorithm 3: Delete-Min

> Parameter : H

```
minitem <- H.data[1]
temp <- H.data[size--]
p <- 1
while p*2 < heapsize
    c <- p*2
    choose a child with smaller weight
    percolate up
H.data[p] <- temp
return minitem
```

Algorithm 4: Weighted-Path-Length

> Parameter: H depth

```
if (no right child)
    then return weight*depth
else
    return sum of recursion(H's left/right child with depth+1)
```

2.2.4 Verification

In this part, we are going to test every submission for its length and ambiguity, if it's optimal then print "Yes", otherwise print "No".

Pseudo-code

Algorithm 1: Overall

```
while M-- > 0
    len <- 0
    for i<-0 to N-1
        INPUT: ch string
        update len with ch_frequency * string.size()
        if (len == length && check(string, N))
            then print("Yes")
        else
            print("No")
```

Algorithm 2: Check

> Parameter: str N

```
sort(str)
for i<-0 to N-1
    for j<-i+1 to N-1
        if (substring(str[j]) is repeated in str[i])
            then return false
return true
```

Chapter 3: Test Result

3.1 Test Case

case	Purposes	Data	Result	Total Time(s)	Run Time	Average Time(s)
1	Sample	data0.txt	ans0.txt	21.681	10000	0.0021681
2	n = 2 (Min), m = 5	data1.txt	ans1.txt	17.477	10000	0.0017477
3	n = 10, m = 11	data2.txt	ans2.txt	18.254	10000	0.0018254
4	n = 20, m = 50	data3.txt	ans3.txt	24.132	10000	0.0024132
5	n = 30, m = 100	data4.txt	ans4.txt	36.72	10000	0.0036720
6	n = 40, m = 200	data5.txt	ans5.txt	6.632	1000	0.0066320
7	n = 50, m = 500(Middle)	data6.txt	ans6.txt	16.466	1000	0.0164660
8	n = 63 (Max), m = 10	data8.txt	ans8.txt	28.476	10000	0.0028476
9	n = 63 (Max), m = 1000 (Max)	data9.txt	ans9.txt	78.69	1000	0.0786900

3.2 Performance

Correctness

The results all meet expectations.

Time Cost

Chapter 4: Analysis and Comments

4.1 Complexity Analysis

Space Complexity:

Since we can process submissions one by one, we have no need to store all m students' answers. Besides this, we use array to be the main container.

- To store input, the space complexity is $O(N)$;
- Our Min-Heap is constructed within an array, which means $O(N)$ space cost ;
- Although Huffman Tree require us to build a tree structure, there are N leaves which means the size of tree is within $k \times N + b$. And the complexity to store Huffman Tree is about $O(N)$;

So the total space complexity of the program is $O(N)$.

Time Complexity:

- In function $CreateHuffmanTree()$ we use heap. The time complexity of Min-Heap operation is $O(N \log_2 N)$;
- To get the right weighted-path-length(WPL), we use function $Wpl()$. And the time complexity is $O(N)$;
- To check whether a submission is legal or not. We need $O(N)$ time to calculate len and check the length correctness. Because the code of characters can not be too long (at most 63 bits), we need $O(N^2)$ time to check if it meets the prefix requirement in other method. Considered that there's m submissions, so the total time cost of examination is $O(m \times N^2)$

So the total time complexity of the program is $O(mN^2)$.

4.2 Comments

With the frequency of different characters, it is easy to get the right WPL in the Huffman tree. Building a Huffman Tree can be completed in $O(N \log_2 N)$ time with heap. But it is even harder to check the prefix requirement which cost $O(mN^2)$ time. Maybe we can use hash or other method to optimize this part.

Chapter 5: Working Arrangements

Programming: 李想

Testing program(main function): 李想

Testing data and plot: 张童晨

Algorithm analysis: 张童晨

Plot comments: 王子腾

Documentation: 王子腾

Declaration

We hereby declare that all the work done in this project titled "Huffman Codes" is of our independent effort as a group.