EE 3980 Algorithms

Homework 9. Encoding ASCII Texts

105061110 周柏宇

2020/5/16

1. Introduction

In this homework, we construct the Huffman code given the paragraph and show the codeword of each character and the ratio of the number of bytes required to store the paragraph using the Huffman code and standard ASCII code.

2. Analysis & Implementation

2.1 Overview

To construct the Huffman encoding table, what we need to do is first obtain the frequency for every character appearing in the paragraph and construct the binary merge tree. Once we obtain the binary merge tree, we can traverse the tree to retrieve the code.

2.2 Obtain Character Frequencies

In this phase, we will go through all characters in a paragraph and accumulate their frequencies, which involves a lot of searching the frequency of a character and increasing them by one. Therefore, it will be good to use a binary search tree to store the characters using their ASCII code as a comparison key. Here’s the data structure for this task.

```
// data structure to store the char and its frequency
struct node {
    int ch; // character
    int n_ch; // frequency
    struct node *l; // left child
    struct node *r; // right child
};
```

The algorithm used to get the character frequencies is as below.

```
// Calculate character frequency
// Input: a paragraph P, binary search tree root bst
// Output: characters and their frequency in a binary
//         search tree, number of characters read
Algorithm getFrequency(P, bst)
{
    n_ch := 0; // number of chars read
    ch := next character in P;
    // read the paragraph
      while (ch is not EndOfFile) do {
          n_ch := n_ch + 1; // # of chars read plus 1
          tmp := bst_find(ch); // find the char in bst
          if (tmp = NULL) {
              // add the char if not in bst
              bst_insert(ch);
          }
          else {
              // increase the frequency of the char by 1
              tmp→n_ch := tmp→n_ch + 1;
          }
          ch := next character in P;
      }

      return n_ch;
 }
```

The while loop will execute as many times as the number of characters in the paragraph, denoted n. In the while loop, we always perform a search in the binary search tree, which takes $\mathcal{O}(\lg n)$ on average and $\mathcal{O}(n)$ for the worst case. Besides, we sometimes need to perform insertion to a binary search tree, which has the same average and worst-case time complexity as searching. Therefore, one iteration of the while loop has an average and worst-case time complexity $\mathcal{O}(\lg n)$ and $\mathcal{O}(n)$ respectively, making `getFrequency` an algorithm with time complexity $\mathcal{O}(n \lg n)$ and $\mathcal{O}(n^2)$ for average and worst case.

2.3 Binary Merge Tree Algorithm

In this phase, we already have the character frequencies. Thus, we can perform the Binary Merge Tree algorithm to get the optimal merge order.

```
// Generate binary merge tree from list of n nodes
// which contains the character and its frequency
// Input: int n, list of nodes
// Output: optimal merge order
Algorithm Tree(n, list)
{
    for i := 1 to (n - 1) do {
        pt := new node;
        // find and remove min from list
          pt→lchild := Least(list);
          pt→rchild := Least(list);
          pt→n_ch := (pt→lchild)→n_ch + (pt→rchild)→n_ch;
          Insert(list, pt);
      }
      return Least(list);
 }
```

As we can see, the algorithm above involves finding minimum in a list. An intuitive way of implementing `Least` is to enforce the min heap property to the list. Therefore, retrieve the minimum will simply be an $\mathcal{O}(1)$ operation.

However, our nodes are stored in a binary search tree right now. We need to first transfer all the nodes to an array then enforce the min heap property.

```
// Store the binary search tree in a list.
// Input: current node node, list arr, index i
// Output: index i
// Initiate the call with
//        bst_to_array(bst, arr, 1)
Algorithm bst_to_array(node, arr, i)
{
    if (node != NULL) {
        arr[i] := node; // add the node to array
          i := i + 1;
          // travel to left child
          i := bst_to_array(node→l, arr, i);
          // travel to right child
          i := bst_to_array(node→r, arr, i);
         // become a leaf node for merging the tree later
          node→l := NULL;
          node→r := NULL;
      }
      return i;
 }
```

```
// Make the array a min heap.
// Input: array A with n elements
// Output: array A with min heap property
Algorithm array_to_minHeap(A, n)
{
    // Enforce min heap property to non leaf node
    // in bottom-up order.
    for i := ⌊n / 2⌋ to 1 step -1 do
        minHeapify(A, i, n);
 }
```

In `bst_to_array`, it is a basic tree traversal operation; hence it has $\mathcal{O}(N)$ time complexity, where N denotes the number of nodes in the binary search tree, i.e. the number of unique characters in the paragraph.

In `array_to_minHeap`, it is exactly the same as the first loop of heap sort except that heap sort uses `maxHeapify`. Therefore, this function has $\mathcal{O}(N \lg N)$ time complexity.

With our list possessing min heap property, the function `Least` and `Insert` in the algorithm `Tree` should be implemented as follows respectively:

```
// Remove minimum from the min heap.
// Input: min heap in an array A with n elements
// Output: minimum of the min heap
Algorithm minHeapRemoveMin(A, n)
{
    if (A = NULL) return NULL; // empty heap
    min := A[1]; // minimum is the root
    A[1] := A[n]; // move last node to root
    minHeapify(A, 1, n - 1); // recover min heap
    return min;
}
```

```
// Insert a node to min heap.
// Input: min heap in an array A with n elements
//        new node to be inserted nn
// Output: none
Algorithm minHeapInsertion(A, n, nn)
{
    i := n; // start at last node
    A[n] := nn; // put new node at last node
    while ((i > 1) and (A[⌊i / 2⌋]→n_ch > nn→n_ch)) do {
          A[i] := A[⌊i / 2⌋]; // parent should be larger
          i := ⌊i / 2⌋; // move up one layer
      }
      A[i] := nn; // put new node at proper place
 }
```

In `minHeapRemoveMin`, although retrieving the minimum is a constant time operation, we still need to restore the min heap property since we will reuse the min heap. Therefore, the time complexity is dominated by `minHeapify`, which is $\mathcal{O}(\lg N)$.

In `minHeapInsertion`, the while loop will execute at most $\mathcal{O}(\lg N)$ since the index is divided by 2 every time, making the insertion an $\mathcal{O}(\lg N)$ operation.

With the above analysis, we can go back to obtain the time complexity of `Tree` $\mathrm{T}_{\mathrm{Tree}}$:

$$
\begin{aligned}
T_{Tree} &= (N-1)\cdot(2\mathrm{T}_{\mathrm{Least}} + \mathrm{T}_{\mathrm{Insert}}) \\
&= (N-1)\cdot\big(2\cdot\mathcal{O}(lgN) + \mathcal{O}(lgN)\big)
\end{aligned}
$$

$$= \mathcal{O}(N\ lgN)$$

Recall that `bst_to_array` has $\mathcal{O}(N)$ and `array_to_minHeap` has $\mathcal{O}(N\ lgN)$ time complexity. Therefore, in this phase, the time complexity is $\mathcal{O}(N\ lgN)$, where N is the number of unique characters and is pretty much a constant for an English paragraph with moderate length.

2.4 Retrieve Huffman Code

Now that we have the binary merge tree that produces the optimal code. What is left to do is to parse the tree and retrieve the code for every character.

```
// Print Huffman code.
// Input: node in the merge tree node,
//        array for the code code
//        i-th bit for the code bit
// Output: none
// Initiate the call with
//   printHuffmanCode(root of merge tree, code, 1, -1)
Algorithm printHuffmanCode(node, code, i, bit)
{
    if (bit != -1) { // not initial call
        code[i - 1] := bit; // set the bit
    }
    if (node→ch != -1) { // leaf nodes
        write(node→ch) // print chars
        // print the code for the char
        for i := 1 to i - 1 do {
            write(code[j]);
        }
    }
    else { // non leaf nodes
        // left child, next bit is 0
        printHuffmanCode(node→l, code, i + 1, 0);
        // right child, next bit is 1
        printHuffmanCode(node→r, code, i + 1, 1);
    }
}
```

`printHuffmanCode` is again a traversal algorithm of the binary merge tree. The fact that the number of nodes in the merge tree is $2N-1$, where N is the number of tree nodes, can be observed by considering that every time we create a new node using two existing nodes, and will not stop if there is more than one node left. Thus, the process will create $N-1$ merged nodes in addition to N leaf nodes. This results a $O(N)$ time complexity.

Besides traversing the tree, when being at leaf nodes, we will print out the corresponding Huffman code, which can be proved to have a length $log_2 \frac{1}{f}$, where f is the normalized frequency of the character. Therefore, to print out all the codes, it will take $\sum_{i=1}^{N} log_2 \frac{1}{f_i}$. Consider the case that f is uniform, i.e., $f_i = \frac{1}{N}$, then $\sum_{i=1}^{N} log_2 \frac{1}{f_i} = N\ log_2 N$. While in general the frequencies we get from a random paragraph will not have a uniformly distributed characters, I think $O(N\ lgN)$ can be our compromised approximation since $\sum_{i=1}^{N} lg \frac{1}{f_i}$ is unbounded.

With the analysis above, we can conclude that the time complexity of `printHuffmanCode` is $O(N) + O(N\ lgN) = O(N\ lgN)$

2.5 Summary

To summarize, we list the time complexity of all phases.

Obtain Character Frequencies: $O(n \lg n)$ (average)

$O(n^2)$ (worst-case)

Binary Merge Tree Algorithm: $O(N\ lgN)$

Retrieve Huffman Code: $O(N\ lgN)$

Generally speaking, $n \gg N$. As a result, the time complexity of the procedure of constructing the Huffman code including collecting the character frequencies is dominated by the first phase.

3. Result and Observation

To observe the relationship between code length `l` and the normalized frequency `p`, I print out some meaningful numeric.

```
Letter   p    log₂(p) l code
----------------------------
' ': 0.17474 2.51670 3 111
e:   0.09047 3.46645 3 001
t:   0.07616 3.71488 4 1100
o:   0.06469 3.95028 4 1000
a:   0.06218 4.00738 4 0111
n:   0.04938 4.34003 4 0101
i:   0.04410 4.50294 4 0001
r:   0.04168 4.58460 4 0000
≈
K:   0.00017 12.54460 13 1010111110100
6:   0.00017 12.54460 13 1010111111100
4:   0.00008 13.54460 13 0100011100110
8:   0.00008 13.54460 13 0100011100111
$:   0.00008 13.54460 14 10101111101010
U:   0.00008 13.54460 14 10101111101011
Number of Chars read: 11949
  Huffman Coding needs 53830 bits, 6729 bytes
  Ratio = 56.3143 %
```

We can see that the code length is bounded by $\log_2 \frac{1}{\mathrm{p}} + 1$, which is the optimal solution for minimizing the average code length $\sum_{i=1}^{N} p_i l_i$, subject to the Kraft Inequality $\sum_{i=1}^{N} 2^{-l_i} \leq 1$. Kraft Inequality is the necessary condition for a binary prefix code e.g. Huffman code. It is good to have the constraint because a prefix code can be decoded without reference to future codewords.