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.

```
1. // Generate binary merge tree from list of n nodes
2. // which contains the character and its frequency
3. // Input: int n, list of nodes
4. // Output: optimal merge order
5. Algorithm Tree(n, list)
6. {
7.     for i := 1 to (n - 1) do {
8.         pt := new node;
9.         // find and remove min from list
10.          pt→lchild := Least(list);
11.          pt→rchild := Least(list);
12.          pt→n_ch := (pt→lchild)→n_ch + (pt→rchild)→n_ch;
13.          Insert(list, pt);
14.     }
15.     return Least(list);
16. }
```

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` T_{Tree}:

$$T_{Tree} = (N-1) \cdot (2\mathrm{T}_{\text{Least}} + \mathrm{T}_{\text{Insert}})$$

$$= (N-1) \cdot \left(2 \cdot \mathcal{O}(lgN) + \mathcal{O}(lgN)\right)$$

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

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.

```
1. // Print Huffman code.
2. // Input: node in the merge tree node,
3. //        array for the code code
4. //        i-th bit for the code bit
5. // Output: none
6. // Initiate the call with
7. //   printHuffmanCode(root of merge tree, code, 1, -1)
8. Algorithm printHuffmanCode(node, code, i, bit)
9. {
10.     if (bit != -1) { // not initial call
11.         code[i - 1] := bit; // set the bit
12.     }
13.     if (node→ch != -1) { // leaf nodes
14.         write(node→ch) // print chars
15.         // print the code for the char
16.         for i := 1 to i - 1 do {
17.             write(code[j]);
18.         }
19.     }
20.     else { // non leaf nodes
21.         // left child, next bit is 0
22.         printHuffmanCode(node→l, code, i + 1, 0);
23.         // right child, next bit is 1
24.         printHuffmanCode(node→r, code, i + 1, 1);
25.     }
26. }
```

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

2.5 Summary

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

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

$\mathcal{O}(n^2)$ (worst-case)

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

Retrieve Huffman Code: $\mathcal{O}(N\ lgN)$

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.

hw09.c

```c
// EE3980 HW09 Encoding ASCII Texts
// 105061110, 周柏宇
// 2020/05/15

#include <stdio.h>
#include <stdlib.h>
#define b2B(x) (x % 8 ? x / 8 + 1 : x / 8) // convert bits to bytes needed

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

NODE *bst = NULL; // binary search tree (bst) root
NODE **minHeap; // min heap
int n_node = 0; // number of nodes in bst
int bst_idx = 0; // index
int n_heap; // number of nodes in heap
int *code; // array for Huffman code
int n_bit = 0; // number of bits needed using Huffman code

int getFrequency(void); // calculate character frequency
NODE *bst_find(char ch); // find the node in bst
void bst_insert(char ch); // insert the node in bst
void bst_to_array(NODE *node); // store the bst in an array
void minHeapify(NODE **list, int i, int n); // enforce min heap property
void array_to_minHeap(NODE **list, int n); // make the array a min heap
NODE *minHeapRemoveMin(NODE **list, int n); // remove minimum from the min heap
void minHeapInsertion(NODE **list, int n, NODE *new_node);
                                                    // insert a node to min heap
void Tree(void); // construct the merged tree for Huffman code
void printHuffmanCode(NODE *node, int i, int bit); // print Huffman code
void freeHeap(NODE *node); // free allocated memory for nodes in a heap

int main(void)
{
    int n_ch; // number of chars read

    n_ch = getFrequency(); // calculate character frequency
    bst_to_array(bst); // store the bst in an array
    array_to_minHeap(minHeap, n_node); // make the array a min heap
    Tree(); // construct the merged tree for Huffman code
    printHuffmanCode(minHeap[0], 0, -1); // print Huffman code
    printf("Number of Chars read: %d\n", n_ch);
    printf("  Huffman Coding needs %d bits, %d bytes\n", n_bit, b2B(n_bit));
```

```
49      // print the ratio of bytes needed with and without using Huffman code
50      printf("  Ratio = %.4f %%\n", b2B(n_bit) * 100.0 / n_ch );
        printf("  Ratio = %.4f %%\n", b2B(n_bit) * 100.0 / n_ch);
51
52      free(code); // free allocated memory storing the Huffman code
53      freeHeap(minHeap[0]); // free allocated memory for nodes in a heap
54      free(minHeap);
55
56      return 0;
57  }
58
59  int getFrequency(void) // calculate character frequency
60  {
61      char ch;
62      int n_ch = 0; // number of chars read
63      NODE *tmp;
64
65      while ((ch = getchar()) != EOF) { // read the paragraph
66          n_ch++; // number of chars read increase by one
67          tmp = bst_find(ch); // find the char in bst
68          if (tmp == NULL) bst_insert(ch); // add the char if not in bst
69          else tmp->n_ch++; // increase the frequency of the char by one
70      }
71
72      return n_ch;
73  }
74
75  NODE *bst_find(char ch) // find the node in bst
76  {
77      NODE *tmp = bst; // initialize it as tree root
78
79      while (tmp != NULL) {
80          if (ch == tmp->ch) return tmp; // found and return
81          // not found, travel to proper child
82          else if (ch < tmp->ch) tmp = tmp->l;
83          else tmp = tmp->r;
84      }
85      return NULL; // node not found
86  }
87
88  void bst_insert(char ch) // insert the node in bst
89  {
90      NODE *tmp = bst; // initialize as tree root
91      NODE *p = NULL; // parent of a node
92      NODE *new_node;
93
94      n_node++; // number of nodes increase by one
95      new_node = (NODE *)malloc(sizeof(NODE)); // allocate memory
96      new_node->ch = ch;
97      new_node->n_ch = 1; // initialize the frequency as one
```

```
98      new_node->l = NULL;
99      new_node->r = NULL;
100     while (tmp != NULL) {
101         p = tmp; // store the parent
102         // travel to proper child
103         if (ch < tmp->ch) tmp = tmp->l;
104         else tmp = tmp->r;
105     }
106     if (p == NULL) bst = new_node; // tree is empty
107     else if (ch < p->ch) p->l = new_node; // smaller than parent
108     else p->r = new_node; // larger than parent
109 }
110
111 void bst_to_array(NODE *node) // store the bst in an array
112 {
113     if (node == bst) { // if node is the root
114         minHeap = (NODE **)malloc(sizeof(NODE *) * n_node); // allocate memory
115         bst_idx = 0; // initialize the index
116     }
117     if (node != NULL) {
118         minHeap[bst_idx++] = node; // add the node to array
119         bst_to_array(node->l); // travel to left child
120         bst_to_array(node->r); // travel to right child
121         // become a leaf node for merging the tree later
122         node->l = NULL;
123         node->r = NULL;
124     }
125 }
126
127 void minHeapify(NODE **list, int i, int n) // enforce min heap property
128 {
129     int j; // index
130     int done; // loop flag
131     NODE *tmp;
132
133     j = 2 * (i + 1) - 1; // initialize j to be left child of i
134     tmp = list[i]; // copy root element
135     done = 0;
136     while ((j <= n - 1) && (!done)) {
137         // let list[j] to be the smaller child
138         if ((j < n - 1) && (list[j]->n_ch > list[j + 1]->n_ch)) j++;
139         if (tmp->n_ch < list[j]->n_ch) done = 1; // exit if root is smaller
140         else {
141             list[(j + 1) / 2 - 1] = list[j]; // replace j's parent with list[j]
142             j = 2 * (j + 1) - 1; // move j to its left child
143         }
144     }
145     list[(j + 1) / 2 - 1] = tmp; // move original root to proper place
146 }
147
```

```
148
149 void array_to_minHeap(NODE **list, int n) // make the array a min heap
150 {
151     int i; // index
152
153     // enforce min heap property to non leaf node in bottom-up order
154     for (i = n / 2 - 1; i >= 0; i--) minHeapify(list, i, n);
155 }
156
157 NODE *minHeapRemoveMin(NODE **list, int n) // remove minimum from the min heap
158 {
159     NODE *tmp;
160
161     if (list == NULL) return NULL; // empty heap
162     tmp = list[0]; // minimum is the root
163     list[0] = list[n - 1]; // move last node to root
164     minHeapify(list, 0, n - 1); // restore min heap property
165
166     return tmp;
167 }
168
169 // insert a node to min heap
170 void minHeapInsertion(NODE **list, int n, NODE *new_node)
171 {
172     int i; // index
173
174     i = n - 1; // start at last node
175     list[n - 1] = new_node; // put new node at last node
176     while ((i > 0) && (list[(i + 1) / 2 - 1]->n_ch > new_node->n_ch)) {
177         list[i] = list[(i + 1) / 2 - 1]; // overwrite list[i] with its parent
178         i = (i + 1) / 2 - 1; // move up one layer
179     }
180     list[i] = new_node; // put new node at proper place
181 }
182
183 void Tree(void) // construct the merged tree for Huffman code
184 {
185     NODE *tmp, *tmp2, *new_node;
186
187     n_heap = n_node; // initialize as number of nodes in bst
188     while (n_heap >= 2) { // stop when only one node in heap
189         // select the first and second smallest nodes
190         tmp = minHeapRemoveMin(minHeap, n_heap--);
191         tmp2 = minHeapRemoveMin(minHeap, n_heap--);
192         new_node = (NODE *)malloc(sizeof(NODE)); // allocate memory
193         new_node->ch = -1; // non leaf node
194         new_node->n_ch = tmp->n_ch + tmp2->n_ch; // sum up the frequency
195         new_node->l = tmp; // smaller node goes to left child
196         new_node->r = tmp2; // larger node goes to right child
197         // insert the merged node back to heap
```

```
198         minHeapInsertion(minHeap, ++n_heap, new_node);
199     }
200 }
201
202 void printHuffmanCode(NODE *node, int i, int bit) // print Huffman code
203 {
204     int j; // index
205
206     if (bit == -1) { // initial call
207         code = (int *)malloc(sizeof(int) * (n_node - 1)); // allocate memory
208         printf("Huffman coding:\n");
209     }
210     else code[i - 1] = bit; // set the bit
211
212     if (node->ch != -1) { // leaf nodes
213         // print chars
214         if (node->ch == 32) printf("\' \': ");
215         else if (node->ch == 10) printf("\'\\n\': ");
216         else printf("%c: ", node->ch);
217         // print the code for the char
218         for (j = 0; j < i; j++) printf("%d", code[j]);
219         printf("\n");
220         n_bit += node->n_ch * i; // accumulate number of bits needed
221     }
222     else { // non leaf nodes
223         printHuffmanCode(node->l, i + 1, 0); // left child, next bit is 0
224         printHuffmanCode(node->r, i + 1, 1); // right child, next bit is 1
225     }
226 }
227
228 void freeHeap(NODE *node) // free allocated memory of nodes in a heap
229 {
230     // post-order traversal
231     if (node != NULL) {
232         freeHeap(node->l);
233         freeHeap(node->r);
234         free(node);
235     }
236 }
```

[Program Format] can be improved.
[CPU time] 0.214119 sec
[Program] output should conform to the example.
[Approach] in constructing A and F can be more efficient.
[Good] effort in writing the report.

Score: 96