EE3980 Algorithms

HW7 Huffman Code

104061212 馮立俞

2018/4/26

Introduction

ASCII character encoding is one commonly-used English encoding system which requires 8 bits (i.e. 1 Byte) for each character. However, it's not very efficient in space usage since not every character appear equally often. In fact, we could compress space usage by encoding frequently-used characters with shorter bits, and less-frequently-used with longer bits. Such encoding method is called Huffman Coding.

Approach

The target of Huffman Coding is minimizing total bit usage, $\sum_{i=1}^{n} b_i * f_i$, where b_i and f_i are bits required to encode and the usage frequency of that character. From Algorithm class we know that a binary merge tree could minimize $\sum_{i=1}^{n} d_i * f_i$, where d_i and f_i are depth of a node and the node's frequency. Therefore, we could achieve Huffman coding by building a binary merge tree by relating bit length with its level in tree.

Building Binary Merge Tree

```
Algorithm Tree(n, list) // Generate binary merge tree from list of n files.
 {
        for i: = 1 to(n- 1) do {
            pt: = new node;pt - > lchild: = Least(list);
                   // Find and remove min from list.
            pt - > rchild: = Least(list);
            pt - > w: =
              (pt - > lchild) - > w + (pt - > rchild) - > w;Insert(list, pt);
            }
        return Least(list);
 }
```

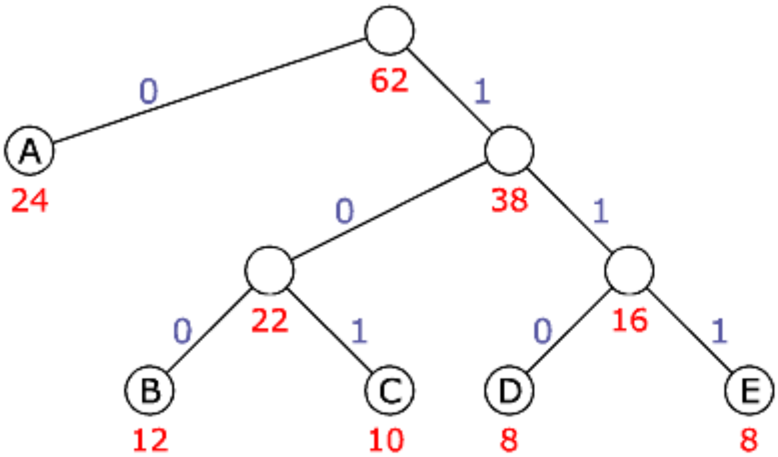
To build a binary merge tree, we keep finding the smallest element in `list`, merging them to form a subtree, then insert the subtree back to `list`, the procedure continues till there's only one element (i.e. root) in `list`.

Depending on how we achieve `Least` and `Insert` function, the resulting tree could be different. I adopted a Min-Heap in this assignment. Due to its unstable sorting property, the nodes with same frequency may switch places, compared to stable sorts. However, the total bit usage isn't affected by how we perform `Least` and `Insert` function. After all, their frequencies are the same.

`Least` and `Insert` function would also affect the complexity of this algorithm. Here because I used Heap, the overall time complexity is $O(nlogn)$, space complexity being $O(n)$ for a Heap and a Tree.

Encoding

After the tree is built, we encode the left child node with additional ‘0’, and the right child with additional ‘1’, like the following figure shows. This operation can be done with recursion call. In practice, I also record bit usage when encoding.




```
Algorithm int Encode(Node * node, char * str) {
    bitCount: = 0;node - > HMcode = str;
    if (is leaf node) { //print leaf nodes
        print( node - > c, node - > HMcode);
        bitCount = strlen(node - > HMcode) * node - > freq; //record freq
    }
    if (node - > lchild != NULL) {
        temp = str + '0';
        bitCount += Encode(node - > lchild, temp);
    }
    if (node - > rchild != NULL) {
        temp = str + '1';
        bitCount += Encode(node - > rchild, temp);
    }
    return bitCount;
}
```

The above recursion call needs to traverse all nodes and edges, whose time complexity is thus $O(n+e)$, n, e are the number of nodes and edges in the tree, respectively. Additionally, the recursive nature would require $O(l)$ space complexity when calling themselves.

Results and analysis

In the given test case, Huffman Code proved to compress bit usage down to around 53% of ASCII version. It seems that the test cases have similar character distribution.

Observations and Conclusion

1.Huffman Code compresses space while preserving some readability.

Intuitively, one might consider another encoding method. That is, '0' for the most common character, '1' for the second, '00' for the third, '01' for the fourth…etc.

Huffman is not optimal?

Though this will save more bits than Huffman code, but it will cause some difficulty decoding it. For example, is '00' representing one or two character, or it's just a fraction of an encoded character? Yet to decode Huffman code, one only need to follow the binary merge tree. And the confusion above can be avoided if there's no noise during message transmission.

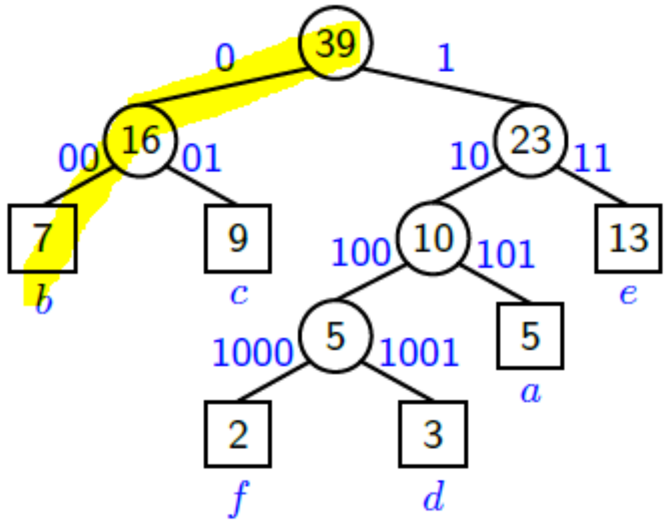


Figure. One only need to follow the tree to decode, say '00'

2.Depending on the adopted sorting method, the resulting Huffman code may not be unique.

```c
/************************************
  EE3980 Algorithms HW07 Huffman Code
  Li-Yu Feng 104061212
  Date:2018/4/23
*************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <stdbool.h>
#include <math.h>

#define LEN 14

typedef struct node{                //node in MinHeap & Tree
    struct node *lchild,*rchild;
    char c, *HMcode;
    int freq;
}Node;


double GetTime(void);
void Heapify(Node *list, int i, int n);     // build a heap for nlgn speed-
void MinHeap(Node *list,int n);             // finding minimal member
Node *HeapPop(Node *list, int n);           //return minimum
void HeapInsert(Node *list, int n, Node *new);  //insert new member to MinHeap
Node *Tree(Node *list, int n);      //build minimum cost tree
int Encode(Node *node, char *str);  // Huffman encoder, print using recursion

double GetTime(void)
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec+1e-6*tv.tv_usec;
}

void Heapify(Node *list, int i, int n){
    int j = i*2;
    Node temp = list[i-1];
    bool done = false;

    while(j<=n && !done){
        if(j<n && list[j-1].freq > list[ j+1 -1].freq ) j++;           //list[j-1].freq > list[ j+1 -1].freq
        if(temp.freq < list[j-1].freq ) done = true;
        else{
            list[j/2-1] = list[j-1];
            j *= 2;
        }
```

```
        //printf("%d\n",j);
    }
    list[j/2-1] = temp;
}

void MinHeap(Node *list,int n){
    Node temp;
    int i;

    for( i = n/2 ; i>0 ; i--)
        {Heapify(list,i,n);}
}

Node *HeapPop(Node *list, int n){
    Node *min;

    min = malloc(sizeof(Node));
    *min = list[0];
    list[0] = list[n-1];
    Heapify(list,1, n-1);   //maintain MinHeap

    return min;
}

void HeapInsert(Node *list, int n, Node *new){      //n = current member count
    int j = n+1;

    while( j > 1 && new->freq < list[ j/2 -1].freq ){                 //new->freq < list[ j/2 -1].freq
        list[j-1] = list[j/2 -1];
        j /= 2 ;
    }
    list[j-1] = *new;

}

Node *Tree(Node *list, int n){
    Node *pt,*temp;
    int i;

    for( i = n; i > 1; ){

        pt = malloc(sizeof(Node));
        pt->c = '.';

        for(temp = HeapPop(list, i--);                      //ignore node with
             temp->freq ==0; temp = HeapPop(list, i--) );   //zero freq
        pt->lchild = temp;
        pt->rchild = HeapPop(list, i--);
```

```

        pt->freq = pt->lchild->freq + pt->rchild->freq;

        HeapInsert(list, i++, pt);
        //printf("Take %c(%d) %c(%d)\n",pt->lchild->c,pt->lchild->freq,pt->rchild->c,pt->rchild->freq );
        //printf("I am %d\n",pt->freq );
    }
    return HeapPop(list,1);
}

int Encode(Node *node, char *str){
    char *temp;
    int bitCount = 0;

    node->HMcode = malloc( sizeof(str) );
    temp = malloc( sizeof(str) +1 );
    node->HMcode = str;
    if(node->c != '.' && node->freq != 0){      //print leaf nodes
        if(node->c == '\0') printf("  \\n:  %s\n",node->HMcode );
        else    printf("  %c:  %s\n",node->c, node->HMcode );
        bitCount = strlen(node->HMcode) * node->freq;
    }

    strcpy(temp,str);

    if (node->lchild != NULL){
        temp[strlen(temp)] = '0';
        bitCount += Encode(node->lchild, temp);
    }

    if (node->rchild != NULL){
        temp[strlen(temp)-1] = '1';
        bitCount += Encode(node->rchild, temp);
    }
    return bitCount;
}

void InsertionSort(Node *list,int n){
    int i,j;
    Node temp;

    for(j = 1; j < n; j++){
        temp = list[j];
        i = j-1;
        while((i>=0) && temp.freq > list[i].freq ){
            list[i+1] = list[i];
            i--;
        }
        list[i+1] = temp;
```

```c
    }
}




int main(){
    int i,j,k;
    int Nwords,Nchar;
    double t;
    char **words;
    Node *list,*temp;
    bool done;

    int count;

    scanf("%d", &Nwords);


    words = (char**)malloc(Nwords * sizeof(char*));         //
    for(i = 0; i < Nwords; i++)                             //
        words[i] = (char *)malloc( (LEN+1) * sizeof(char)); //


    for(i = 0; i < Nwords ; i++){                           //
        scanf("%s", words[i]);                              //scan words
    }

    list = (Node *)malloc( 27 * sizeof(Node));
    list[0].c = '\0';
    list[0].freq = 0;

    for (i = 1; i < 27; i++){       //init list
        list[i].c = 'a'-1 +i;
        list[i].freq = 0;
    }
    Nchar = 0;
    list[0].freq = Nwords;
    for (i = 0; i < Nwords ; i++){              //count char freq
        for(j = 0; words[i][j] != '\0'; j++)
            list[ words[i][j]-'a'+1 ].freq++;
        Nchar += j+1;
    }

    printf("Number of words: %d\nNumber of characters: %d\n",Nwords,Nchar );

    /*InsertionSort(list,27);
    for ( i = 0,j = 1, k = 2, count = 0; i < 27; ++i)
    {
        if(i+1 <= k)   count += list[i].freq * j;
```

```
        else{
            j++; k *= 2;

            count += list[i].freq * j;
        }
    }
    printf("dummy encoding bit count:%d\n",count );*/




    printf("Huffman coding\n");

    MinHeap(list,27);

    temp = Tree(list,27);
    i = Encode(temp,"");
    printf("Number of encoded bits: %d\n", i );
    printf("Ratio: %.4f%\n", i / (Nchar*8.0)*100.0 );
    return 0;

}
```

Score: 60

[Compression] ratios should be listed in your report.

[Segmentation] fault may appear when running your compiled program.
- node->lchild, node->rchild not properly initialized?

[Compiler warning] line 215.

[Program] can have more comments.

[Algorithms] need to be explained more clearly.

[Report] can be improved.