EE3980 Algorithms

Homework 8. Huffman Code

By 105061212 王家駿

1. Introduction

In order to store characters or strings in the memory, we often use ASCII or UTF-8 code to encode the English letters. However, since it always uses eight bits to encode a letter in ASCII code, it seems as a waste of storage space in the case when the characters repeat many times. Therefore, we use a variable length encoding, where the encoding length depends on the frequency of each character used. And in this homework, we use one of the variable length encoding methods, Huffman Code, to encode the lowercase letters appearing in the file. In the end, we compare the bits used for Huffman Code and those use for traditional ASCII code, and figure out the complexity of encoding.

2. Implementation

For implementation, we first read the input file, and count the number of times each character appears. Then, we use the counting table to encode the letters appearing in the file.

2.1. Method of encode – Huffman Code

The Huffman Code use a binary tree to encoding the letters. From the root, the left child ang right child are encoded as 0 and 1, respectively. Then keep going down until find the leaf node, which is the character must be put in. So, the characters are encoded by 0’s and 1’s, depending on the path (left/right child) to find the exact node, and the length of each code differs, which relates to the number of nodes during traversal. That is, the deeper the character node is, the longer its Huffman Code.

Since we want to minimize the total bits used, the characters with higher frequencies must be encoded as short as possible. Thus, we should put them at the upper area of the tree, while putting the characters with lower frequency deeper. So the structure would be like a binary tree with characters as its leaf nodes and inner nodes as the sum of its children.

2.2. Node strcture

```c
typedef struct node                         // binary merge tree node
{
    struct node* lchild;                    // pointer to left child
    struct node* rchild;                    // pointer to right child
    char c;                                 // character data
    int freq;                               // used frequency
    int leaf;                               // node is leaf or not
} Node;
```

To construct the binary tree, we define the structure of each tree node. The two pointers in the structure point to two tree node as its left child and right child; the character c stores the information of the character the node represents; the integer frequency stands for the total number the character appears; the integer leaf is set to 1 if the node is a leaf node, and 0 if it is not.

2.3. Binary merge tree (BMT)

```
Algorithm BMT(n, list, tree)
{
    for i := n to 2 step -1 do {
        t := new node;
        t.lchild := Least(list);      // Find and remove min from list
        t.rchild := Least(list);      // Find and remove second min from list
        t.freq := (t.lchild).freq + (t.rchild).freq;
        Insert(list, t);
        Insert(tree, t.lchild, t.rchild);   // construct tree
    }
    Insert(tree, list[1]);
}
```

We use the binary merge tree (BMT) algorithm to construct the binary tree for Huffman Code.

In the BMT algorithm, first we get an array with frequencies to be constructed to a tree. At each iteration step, we picked up the smallest and the second smallest node in the array, and then allocate a new node, whose left child and right child are the two smallest nodes. Thus, the frequency of the new node would be the sum of its children's, and it is reasonable since it's as same as the sum of times of the two children's character appear. Then, we put the children node into the tree structure since they are well arranged by their parent's pointer, and put the parent node into the list for the next iteration. At the end of the iteration, all nodes except list[1] are placed into the tree. So, we finally put the node as the root of the tree and the whole binary tree is constructed.

For any node in the binary tree, the frequency of the parent must be equal or greater than its children's since $\mathrm{f(parent)} = \mathrm{f(lchild)} + \mathrm{f(rchild)}$. Thus, the binary tree is a max heap, with the larger on the top and the smaller at the bottom. Moreover, at each iteration step, we always pick the smallest two elements, so we could ensure that they would be put at the lower place. Above all, they are same as the requirement of Huffman Code, so we could use the BMT algorithm to solve the Huffman code problem.

2.4. Search for the smallest

At line 5 and 6 of BMT algorithm, we have to find out the smallest item in the list array. For an array with size n, one way to find the smallest is linear search, which means iterating through the whole array and recording the smallest item so far. After the iteration we could find the smallest element, and the time complexity is $\mathrm{O(n)}$.

Another way is to make the list array to be a min heap. Since the property of minheap (the parent node is smaller than its children for all nodes), we could easily get the smallest item, which is list[1], and the time complexity is $O(1)$. Thus, every time we want to get the minimum, we could first adjust the array into min heap, and then take list[1] as the smallest item.

For implementation, we just revise the heap sort algorithm we have done in homework 2. The time complexity is $O(\log n)$.

```
Algorithm Heapify(A, i, n)
{
    j := 2×i; item := A[i]; done := false ;     // A[j] is the lchild
    while ((j ≤ n) and ( not done )) do {       // A[j+1] is the rchild
        if ((j < n) and (A[j] > A[j+ 1])) then
            j := j+ 1;                          // A[j] is the smaller child
        if (item < A[j]) then                   // If smaller than children,
            done := true ;
        else {                                  // Otherwise, continue
            A[j/2] := A[j];
            j := 2×j;
        }
    }
    A[j/2] := item;
}

Algorithm buildMinHeap(n)
{
    for i := n/2 to 1 step -1 do      // for nodes with children, heapify
        Heapify(A, i, n);
}
```

Thus, we could implement heapify every time when we try to get the smallest element. According to the property of min heap, the second smallest item must be one of the child of the root, list[2] or list[3]. Thus, we could get the two smallest element by using heapify once and lead to an $O(\log n)$ time complexity.

For the BMT algorithm, the insert statements take constant time, so the overall time complexity is dominated by the loop at line 3 and the search at line 5 and 6. Thus, the time complexity is $O(n) * O(\log n) = O(n * \log n)$, where n is the problem size of characters to be encoded. In this homework, it is the lowercase English letters and '\n', so n = 27.

2.5. Result output

After constructing the binary tree, the last problem is how to traversal the tree the get the coding. We start from the root, and implement depth first search (DFS) to traversal the whole tree, and get the encoding numbers.

```
Algorithm DFS(v)
{
    if (v is leaf node) then            // character node
        print out the stack content;
    else {
        stack.push(0);                  // encode 0
        DFS(v.lchild);                  // keep finding left child
        stack.pop();

        stack.push(1);                  // encode 1
        DFS(v.rchild);                  // keep finding right child
        stack.pop();
    }
}
```

In the DFS, we use a stack to record the path we went through, 0 for the left child and 1 for the right child. When entering a new node, push a bit to the stack and pop it when leaving. Once we get to the leaf node, i.e. the node with character data, we record the bit value in the stack, and that's the Huffman Code.

The traversal would visit every node one time, so the time complexity is $O(\text{number of nodes}) = O(n + (n - 1)) = O(n)$ since there are n-1 inner node in the tree.

3. Executing results

We run the testing data from wl1.dat to wl9.dat with different input data size, and record the bits used comparing to the ASCII code.

Number of characters	**Number of encoded bits**	**Ratio (Huffman/ASCII)**
376	1600	53.19%
722	3086	53.43%
1453	6212	53.44%
3008	12846	53.38%
5972	25548	53.47%
11808	50592	53.56%
23912	102329	53.49%
47246	201929	53.42%
94661	405029	53.48%

4. Result analysis and conclusion

From the measured results we could find that there’s a great reduction on the memory used comparing to the traditional ASCII code. From our testing data, the space that Huffman Code uses is about 53% of ASCII code, roughly half of it. Thus, it is useful for storing a large scale of English words, and it needs about 53% space comparing to the ASCII for general data input.

We had known that the time complexity for constructing the Hoffman binary tree is $O(\mathrm{n} * \log n)$. However, n here represents the different characters to be encoded, which is 27 in this homework. Thus, the n is so small that the complexity would easily be affected by other statement, which may be have the lower time complexity but larger executing time. So, in this case, perhaps the linear search may be faster than heapify in reality.

As for the larger n, the algorithm would have a trend of $O(\mathrm{n} * \log n)$. However, it might be a waste of time of construct the binary tree for encoding because there’s no need for other operations for ASCII code. Therefore, it might become a drawback of Huffman Code. If there has a limit of space during data transmission, and doesn’t need to consider about the encoding procedure, the algorithm is still useful for information interchange.