EE 3980 Algorithms

Term Project. Encoding utf-8 Files

105061110 周柏宇

2020/6/20

1. Introduction

In this homework, we generate the codes given a utf-8-encoded paragraph to minimize the storage space, and show the codeword of each symbol and the ratio of the number of bytes required to store the paragraph using the proposed code and standard utf-8 encoding.

2. Analysis & Implementation

2.1 Overview

For this task, I used the Huffman code to encode every utf-8 symbol that appears. According to the format of utf-8, a symbol can take from 1 to 4 bytes, using a specific pattern of bits to differentiate the types. Therefore, based on byte 1, we can expect how many following bytes needed to compose a symbol.

Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4
1	0xxx xxxx	-	-	-
2	110x xxxx	10xx xxxx	-	-

3	1110 xxxx	10xx xxxx	10xx xxxx	-
4	1111 0xxx	10xx xxxx	10xx xxxx	10xx xxxx

2.2 Obtain Character Frequencies

In this phase, we will go through all symbols in a paragraph and accumulate their appearance, which involves frequent searching. Therefore, it will be ideal to use a hash function to put them into different buckets, and in a bucket, we can use linked lists to connect blocks of information. Here’s the data structure for the information of symbols.

```c
// data structure to store the char and its frequency
struct node {
    unsigned int val; // symbol hash value
    int freq; // frequency
    char *symbol; // utf-8 symbol
    struct node *l; // left child
    struct node *r; // right child
};
```

The left, right child is for later purpose of converting them into a binary merge tree. In this phase, we connect them only using left child to make it function like a linked list. One might ask, why not in every bucket, you store them into a binary search tree instead of a linked list? Because in my implementation, I use 1000 buckets and thus the average number of nodes in a bucket is quite small. On the other hand, inserting a new node in the head of linked list is $O(1)$; while inserting a node to a binary search tree is $O(\lg n)$. After some experiments, it shows that using liked list is faster in practice.

As for the hash function, an intuitive way of doing this is concatenating all the bytes in a utf-8 symbol to be its hash value. There are two advantages. One, it is fast and it will take no more than 4 bytes, which fits perfectly in an unsigned integer. Second, the hash value has a one-to-one correspondence to the symbol, which guarantees no collision of hash values will occur. We put them into the corresponding bucket by taking $hash\_value \bmod number\_of\_buckets$.

```
// Return hash value of a symbol.
// Input: a utf-8 symbol
// Output: 4-byte hash value
unsigned int hash(char *str)
{
    i := 0;
    hash := 0;
    while (str[i] != '\0') do {
          hash := (hash << 8) + str[i];
          i := i + 1;
     }
     return hash;
 }
```

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

```
// Accumulate symbol frequency of a utf-8
// encoded paragraph.
// Input: utf-8 encoded paragraph P
//        array of linked list head bucket
// Output: number of bytes read n_byte
Algorithm readText(P, bucket)
{
  n_byte := 0;
  symbol[1] = '\0';
  ch := first byte in P;
  while (ch is not EndOfFile) do {
    n_byte := n_byte + 1;
    type := det_nB(ch); // determine n-byte symbol
    if (type > 0) { // skip byte 10xx xxxx
      if (symbol[1] != '\0') {
        val := hash(symbol);
        tmp := find(bucket, val);
        if (tmp = NULL) {
          insert(bucket, val, symbol, type);
        }
        else tmp->freq := tmp->freq + 1;
      }
      i := 1;
      symbol[type + 1] = '\0';
    }
    symbol[i] := ch;
    i := i + 1;
    ch := next byte in P;
  }
  val := hash(symbol);
  tmp := find(val);
  if (tmp = NULL) insert(bucekt, val, symbol, type);
  else tmp->freq := tmp->freq + 1;
  return n_byte;
}
```

The while loop will execute as many times as the number of bytes in the paragraph, denoted n_byte. However, there are only n_S times we perform search or insertion, where n_S denotes the total number of utf-8 symbols in the paragraph.

In the while loop, we sometimes execute `find` to search the symbol in the linked list, which takes $O(n\_uS)$ for the worst case (if we use n buckets, then the expected coefficient will be $\frac{1}{n}$, as n being sufficiently large, $\frac{1}{n} \cdot n\_uS$ will be really small), where n_uS denotes the number of unique utf-8 symbols. Besides, we also need to perform `insert` to add a new symbol to the head of a linked list, which takes $O(1)$. Therefore, one iteration of the while loop has a worst-case time complexity $O(n\_uS)$, making `readText` an algorithm with theoretic worst-case time complexity

$$O(n\_byte + n\_S \cdot n\_uS) \approx O(n\_S \cdot n\_uS) \approx O(n\_S^2).$$

2.3 Binary Merge Tree Algorithm

Once we acquire the frequency of symbols, we can now perform the Binary Merge Tree algorithm to obtain the optimal merge order.

```
// Generate binary merge tree from list of n nodes
// which contains the symbol 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→l := Least(list);
    pt→r := Least(list);
    pt→freq := (pt→l)→freq + (pt→r)→freq;
    Insert(list, pt);
  }
  return Least(list);
}
```

As we can see, the algorithm above involves finding the minimum in a list. One way of implementing `Least` is to enforce the min heap property to the list. Therefore, retrieving the minimum will simply be an $\mathcal{O}(1)$ operation and maintaining the min heap property takes $\mathcal{O}(\lg n)$, where n is the number of nodes in the list. Before calling `Tree`, we need to first transfer all the nodes from the linked list to an array then enforce the min heap property.

```
// Transfer all nodes in linked list to an
// array and enforce min heap property.
// Input: array of head of linked list bucket
//        total number of nodes n
//        number of buckets N
//        array with n space minHeap
// Output: min heap in an array
//         using frequency as key
Algorithm hash2minHeap(bucket, n, N, minHeap)
{
    j := 1;
    for i := 1 to N do {
        tmp := bucket[i];
        while (tmp != NULL) do {
            minHeap[j] := tmp;
            j := j + 1;
            tmp := tmp→l;
        }
    }
    for i := ⌊n / 2⌋ to 1 step -1 do
        minHeapify(minHeap, i, n);
}
```

In `hash2minHeap`, the first loop transfers all nodes to an array, which takes $O(n\_bucket + n\_uS) \approx O(n\_uS)$ time complexity. Recall that n_uS denoted the number of unique symbols. For the second loop, it is exactly the same as the first loop of heap sort except that heap sort uses `maxHeapify`. Therefore, this function has $O(n\_uS \cdot \lg n\_uS)$ 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⌋]→freq > nn→freq)) 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 an $\mathcal{O}(1)$ 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 $\mathcal{O}(\lg n\_uS)$.

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

With the above analysis, we can go back to obtain the time complexity of Tree T_{Tree}:

$$T_{Tree} = (n\_uS - 1) \cdot (2T_{Least} + T_{Insert})$$

$$= (n\_uS - 1) \cdot (2 \cdot \mathcal{O}(\lg n\_uS) + \mathcal{O}(\lg n\_uS))$$

$$= \mathcal{O}(n\_uS \bullet \lg n\_uS)$$

Recall that hash2minHeap has $\mathcal{O}(n\_uS \bullet \lg n\_uS)$ time complexity. Therefore, in this phase, the time complexity is $\mathcal{O}(n\_uS \bullet \lg n\_uS)$, where n_uS is the number of unique symbols.

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 symbol.

```
// 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->val != 0) { // leaf nodes
        write(node->symbol) // print utf-8 symbol
        // print the code for the symbol
        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` traverses a binary merge tree. We have proved that the number of nodes in the binary merge tree is $2N - 1$, where N is the number of leaf nodes and $N = n\_uS$. This results a $O(n\_uS)$ time complexity.

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 symbol. Assuming that $f_i = \frac{1}{n\_uS}$, then the sum of code length for every symbol will be $\sum_{i=1}^{n\_uS} log_2 \frac{1}{f_i} = n\_uS \cdot log_2 n\_uS$. Thus, we take $O(n\_uS \cdot \lg n\_uS)$ as the upper bound of the complexity.

With the analysis above, we can conclude that the time complexity of `printHuffmanCode` is

$$O(n\_uS) + O(n\_uS \cdot \lg n\_uS) = O(n\_uS \cdot \lg n\_uS)$$

2.5 Summary

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

Obtain Symbol Frequencies: $O(n\_S \ \cdot n\_uS) \approx O(n\_S^2)$ (worst-case)

Binary Merge Tree Algorithm: $O(n\_uS \cdot \lg n\_uS)$

Retrieve Huffman Code: $O(n\_uS \cdot \lg n\_uS)$

Where n_S is the number of utf-8 symbols in the paragraph and n_uS is the number of unique utf-8 symbols.

Generally speaking, $n\_S \gg n\_uS$. 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, and that is why we should put more work into designing the data structure.

For the space complexity, we need

$O(N\_bucket + n\_uS) \approx O(n\_uS)$ for the linked list

$\Theta(2 \cdot n\_uS - 1) = \Theta(n\_uS)$ for the binary merge tree

$O(n\_uS)$ for the array to store the Huffman code of a symbol

And other space independent of input sizes.

3. Result and Observation

We test our algorithm on several input files.

File name	*Content*	n_uS	n_S	n_byte	*Ratio* [%]
article1.txt	Chinese, English	1217	6213	17993	37.1089
article2.txt	Chinese	790	7607	22619	31.4382
article3.txt	Chinese	2975	41274	123144	37.9637
article4.txt	Chinese	252	586	1688	31.6351
article5.txt	Chinese	681	3974	11516	34.7256
article6.txt	English, Emoji	120	1180	1353	56.3193
article7.txt	English	70	11949	11949	56.3143

One thing we can observe is that the ratio is especially smaller when encoding paragraphs containing Chinese. It is because that utf-8 uses 3 bytes

(24 bits) to encode a Chinese character while the number of unique Chinese characters has merely a couple thousands in our test files, which is far lesser than 2^{24}. This observation got me thinking: can we achieve a better compression rate by combining multiple bytes into a super-symbol, and as long as the number of bits used to represent a super-symbol is far larger than the number of unique super-symbols, we can probably save some space. To test it, I group 4 bytes into a super-symbol, which generally does not have reasonable meaning in utf-8, and generate the Huffman code.

	n_uS		*Ratio [%]*	
File name	*symbol-wise*	*4-byte-wise*	*symbol-wise*	*4-byte-wise*
article1.txt	1217	3432	37.1089	35.9306
article2.txt	790	3328	31.4382	34.8203
article3.txt	2975	17263	37.9637	41.3784
article4.txt	252	384	31.6351	26.8957
article5.txt	681	2146	34.7256	34.1177
article6.txt	120	306	56.3193	25.8684
article7.txt	70	2037	56.3143	33.5342

The result is surprisingly good, especially for the all-English paragraphs, the number of unique symbols increases, but still far lower than 2^{32}. But one thing, during the experiment, I realize is that it generates a long list of conversion table. If we really want to decode the paragraph, shouldn't we take the size of the table into consideration? One extreme example is that we simply combine all bytes of the paragraph and encode it with one bit. The compression rate is absolutely high. What we kind of end up doing is transferring the information from encoded file to conversion table, and it is far from practical. I am pretty sure this is not what the project meant to be, so I stick with the symbol-wise encoding.