EE3980 Algorithms

Term Project. Encoding utf-8 Files

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing several algorithms. The goal of the algorithms is to compress the input texts into some encoded texts that saves more spaces than before.

During the analysis process, I will first introduce some ways or algorithms to achieve the goal of this project. I will approximately analyze them for their time/space complexities, performance on compressing texts, and difficulties.

The implementation of the algorithm on C code will find encoded texts for texts in 7 files. Furthermore, I will calculate their performance and evaluate their CPU times.

Analysis:

I come up with 4 ways to compress texts. The first one is Huffman encoding, which was learned before. Second, Lempel-Ziv-Welch. Third, run length encoding and the last, natural language processing word embedding.

1. Huffman encoding

a. Analysis

Huffman encoding is a way of representing a set of characters based on their appearing frequencies. The data table will be each existed character and frequency table will store their appearing frequencies. After Sorting characters depending on their frequencies, we implement Binary Merge Tree method to merge the least two elements until there is only one element left, and that will be the root of our Huffman Tree. Finally, we traverse the Huffman Tree to print out our requested encoded text in Huffman Encoding Table. We use Heap sort as the sorting method during Huffman encoding process to achieve O(nlgn) on time complexity.

b. Performance

Using Huffman encoding, we can compress Utf-8 texts by approximately 37% and compress ASCII codes by approximately 56%.

c. Difficulty

Implementing Heap sort and Binary Merge Tree are rather easy on C code.

2. Lempel-Ziv-Welch (LZW)

a. Analysis

The idea of the LZW compression algorithm is the following: as the input data is being processed, a dictionary keeps a correspondence

between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed.

Take article1.txt for an example. The first 5 Utf-8 character is “我所知道的”. So we add “我” to our dictionary first and give it a value. If we set our LZW as 2-gram, then we add “我所” to our dictionary, like fig 1. Then, we add “所” and “所知”, … and so on (fig 2).

Char	val	Char	val
我	1	我所	2

Fig 1

Char	val	Char	val
我	1	我所	2
所	3	所知	4
知	5	知道	6
道	7	道的	8

Fig 2

If we see the same text again, we do not need to add it to our dictionary.

So, the if there is a lot of “知道” in our text file, we can compress 6-char-sized strings to simply 1-char-sized.

If we increase the text to 4-gram, we have to add “我所”, “我所知”, “我所知道” in our dictionary. The next time we see “我所知道”, it can be compressed to simply 1 character.

By this method, the frequently appearing words are compressed to a specific value, therefore, it suits some academic articles and especially

texts that repeats a lot.

The downside for this will be it need a lot space to store the dictionary. The time complexity for this will be O(n) * O(lgn) for building dictionary and searching frequencies plus at least O(nlgn) for searching. The time complexity should be O(nlgn).

b. Performance

In our text files, there are not many repeating words. Therefore, LZW is not suitable for this task. It needs at most 4000 or 5000 values to encode article1.txt, that is to say, it needs to store lots of bits to represents 4000 or 5000 values. Even if we modify the algorithm by assigning less-bit values to more frequently used words, the compressing ratio is not better than Huffman encoding.

c. Difficulty

The implementation of LZW is not hard using C with good data structure. It is rather easier and more direct than Huffman coding.

3. Run length encoding

a. Analysis

Run-length encoding is a way of compressing texts by counting the repeating characters. Since Utf-8 has at most 4 bytes, their length can be

the same by seeing them as 4 bytes. Take “我” as an example, “我” in binary bits is “11100110 10001000 10010001”. We can convert the binary bits to “3*1, 2*0, 2*1, 1*0, 1*1, 3*0, 1*1, 3*0, 1*1, 2*0, 1*1, 3*0, 1*1”. And by storing the first appearing bit and the sequence of repetition of bits as an linked list, we can encode the utf-8 characters.

First bit												
1	3	2	1	1	3	1	3	1	2	1	3	1

This way, originally “我” needs 24 bits to store, now it needs only 19 bits. Therefore, the whole text file can be compressed to a smaller file.

The upside for this is that it does not need to store a dictionary or a binary tree to trace decode the texts.

The time complexity for run length is O(N) for N characters.

b. Performance

Since the bits in Utf-8 characters do not repeat a lot, this encoded method’s performance is not as well as Huffman encoding.

c. Difficulty

The implementation of Run-length encoding is rather easier than all the algorithm above since it only needs to transfer texts.

4. Natural language processing word embedding + Huffman

a. Analysis

Before implementing natural language processing tasks, we have to perform some transforming on raw texts. We compute words into vectors by their similarities. Therefore, when implementing tasks such as word generation or comparison, the more similar word vectors are computed together.

Before word embedding, we have to know which characters form a word. For English, it is easy. However, for Chinese, we may need some help with python. There is a package called Jeiba that can separate Chinese characters into words.

Therefore, I wrote a python code to transform single texts in articles to meaningful words (Fig 3). And then, perform Huffman encoding.

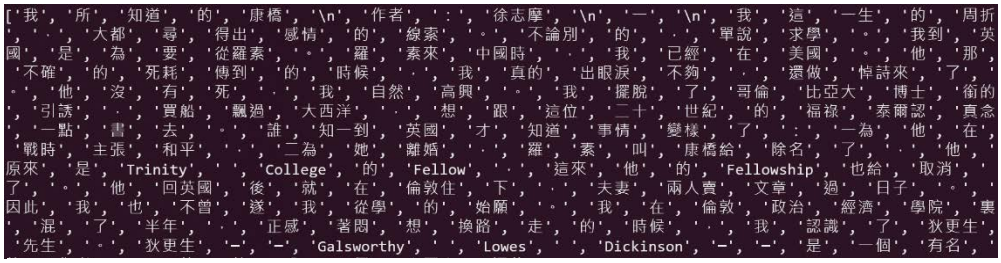


Fig 3

This way, instead of using single Chinese characters as an input for Huffman encoding, using words as input makes Huffman tree smaller.

Fig 4

For English, we can separate words instead of single letters (Fig 4).

After the encoding process (Fig 5), we can calculate the total bits needed (Fig 6). Below is the compression ratio for article1.txt.

Fig 5

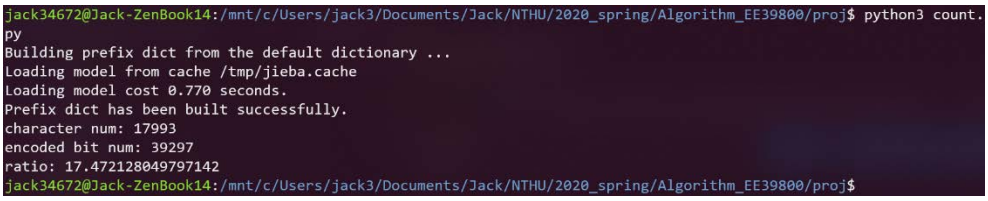


Fig 6

b. Performance

By this method, we can compress all texts by approximately 20%.

c. Difficulty

It can be done by python, but not C.

Implementation:

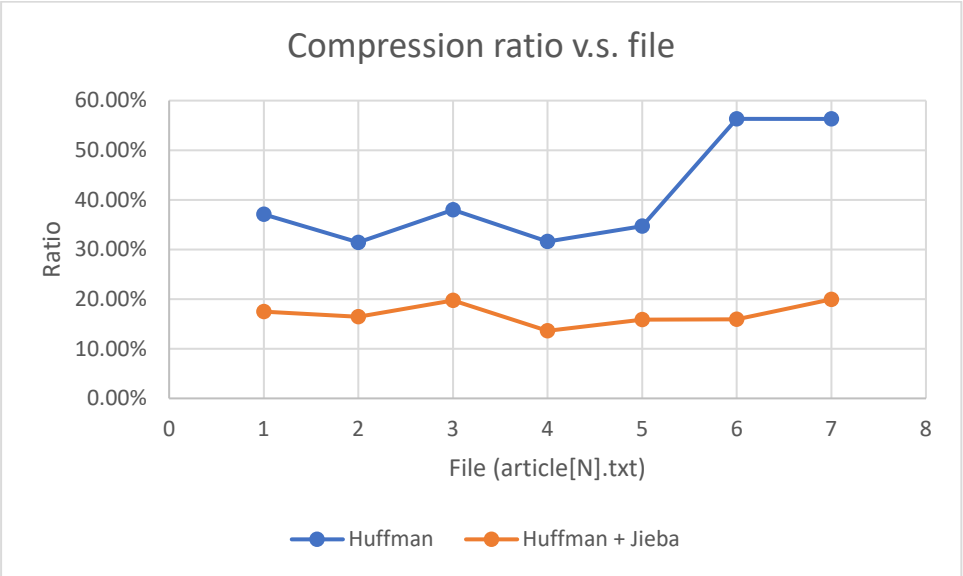
1. Workflow:

```
1.  Algorithm workflow()
2.  {
3.      X = ReadInput();
4.      X = PrepareInput(X);
5.      Y = Encode(X);
6.      Ratio = (Calculate(Y) / Calculate(X));
7.  }
```

2. Result:

Huffman		Jieba + Huffman	
	Ratio		Ratio
article1.txt	37.11%	article1.txt	17.47%
article2.txt	31.44%	article2.txt	16.43%
article3.txt	37.96%	article3.txt	19.76%
article4.txt	31.64%	article4.txt	13.61%
article5.txt	34.73%	article5.txt	15.86%
article6.txt	56.32%	article6.txt	15.95%
article7.txt	56.31%	article7.txt	19.95%

Observation:



We can see that using Huffman encoding, we can compress Chinese text files by approximately 36% and English text files by 55%. By using Huffman encoding plus Jieba word separator, we can compress all files by less than 20%.

The correlation between compression ratios and text files is the repetition of words. The reason of this is because Huffman encoding compression ratio rely on the frequency of the appearing words.

Conclusions:

1. Huffman encoding is still better on encoding texts.
2. Encode words instead of single characters perform better on compression ratio, but it requires Python's help.
3. Result:

Huffman		Jieba + Huffman	
	Ratio		Ratio
article1.txt	37.11%	article1.txt	17.47%
article2.txt	31.44%	article2.txt	16.43%
article3.txt	37.96%	article3.txt	19.76%
article4.txt	31.64%	article4.txt	13.61%
article5.txt	34.73%	article5.txt	15.86%
article6.txt	56.32%	article6.txt	15.95%
article7.txt	56.31%	article7.txt	19.95%

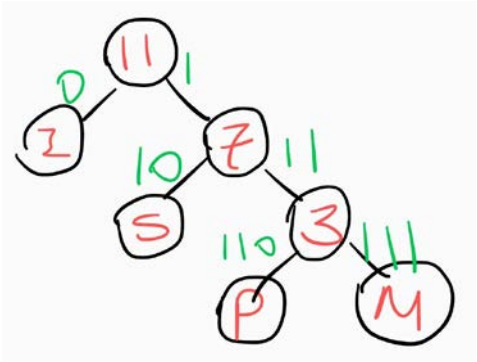
Huffman encoding reference (hw09):

1. Huffman Code:

Huffman encoding is a way of representing a set of characters based on their appearing frequencies. Take the word Mississippi for an example. The data table will be each existed character and frequency table will store their appearing frequencies.

	[0]	[1]	[2]	[3]
data	M	i	s	p
freq	1	4	4	2

After Sorting characters depending on their frequencies, we implement Binary Merge Tree method to merge the least two elements until there is only one element left, and that will be the root of our Huffman Tree. Take Mississippi as an example again. After sorting it into ascending order, we get M and P and merge frequency 2 + 1 = 3 into the tree. By doing so until there is only element 11 left.



Finally, we traverse the Huffman Tree to print out our Huffman Encoding Table.

We use Heap sort as the sorting method during Huffman encoding process since there's sorting and it constantly pops the minimum value. Using heap sort ideally has O(n lg n) and O(1) for the two tasks.

2. Data Structure:

a. Node:

A Node has 4 properties:

properties	definition
Node->(char)data	character

Node->(int)freq	character's frequency
Node->(Node) left	Left node
Node->(Node) right	Right node

b. Heap:

A Heap is to store the sorted character/frequency array and store the merged Binary merge tree.

properties	definition
Heap->(int)size	character
Heap->(Node)*array	Sorted array

3. Heap Sort:

a. Abstract:

Heap sort is a sorting method derives from Heap tree. With heap-representation of the array, heap sort can easily sort input array by *Heapify*, which is an algorithm that maintain heap-property on specific node.

This method of Heapify is slightly different from the original one. It constantly Heapify the child item of the inserted node (i). The process is similar to the old one that it constantly get the heaped first item and Heapify the rest of them.

b. Algorithm:

```
1.  // To enforce min heap property for n-element Heap with root i.
2.  // Input: size n max heap array Heap->array, root i
```

```
// Output: updated Heap.
Algorithm Heapify(list, i, size)
{
    s := i;
    left := 2 * i + 1;
    right := 2 * i + 2;

    if (left < size && list[left]->freq < list[s]->freq) {
        s := left;
    }
    if (right < size && list[right]->freq < list[s]->freq) {
        s := right;
    }
    if (s != i) {
        t := list[s];
        list[s] := list[i];
        list[i] := t;
        Heapify(list, s, size);
    }
}
```

```
Algorithm HeapSort(list, n)
{
    for i := (n - 1) / 2 to 0 step -1 do  {
        Heapify(list, i, size);
    }
}
```

c. Time Complexity:

For the Heapify process, it traverse and heapify until the input node is a leaf node, therefore the time complexity will be O(lg n). For the HeapSort process, it does Heapify for n/2 times, therefore the total time complexity will be n/2 * O(lg n) equals **O(n lg n)**.

d. Space Complexity:

The algorithm uses several integers and Node(t) and a Heap array list[n]. **The space complexity would be O(N).**

4. Binary Merge Tree, BMT():

a. Abstract:

BMT() gets the least two element, which are characters that has least two frequencies, and merge them into one element with frequencies added.

b. Algorithm:

```
1.  // Generate binary merge tree from list of n files.
2.  // Input: int n, list of files
3.  // Output: optimal merge order.
4.  Algorithm BMT(n, list)
5.  {
6.      for n > 0 do {
7.          pt := new node ;
8.          pt -> lchild := GetMin(list) ; // Find and remove min from list.
9.          pt -> rchild := GetMin(list) ;
10.         pt -> w := (pt -> lchild) -> w + (pt -> rchild) -> w ;
11.         while (n && pt -> freq < list[(n - 1) / 2]->freq) {
12.             list[n] =list[(n - 1) / 2];
13.             n := (n - 1) / 2;
14.         }
15.         list[n] := pt;
16.     }
17.     return GetMin(list) ;
18. }
19.
20. Algorithm GetMin(list)
21. {
22.     temp := list[0];
```

```
23.     list[0] := list[size - 1];
24.     size--;
25.     Heapify(list, 0, size);
26.     return temp;
27. }
```

c. Time complexity:

For GetMin(), there is one Heapify in it, therefore it's time complexity will be O(lg n).

For BMT(), there is GetMin and a while loop with O(lg n) frequency (since it iterates with n, n/2 n/4 … etc). The time complexity for BMT() will be **O(n lg n)**.

d. Space Complexity:

The algorithm uses several integers and Nodes(temp) and a Heap array list[n]. **The space complexity would be O(N).**

5. Huffman Tree Traversal, PrintCode():

a. Abstract:

PrintCode() prints each input character's Huffman Code by traversing the Huffman Tree.

b. Algorithm:

```
1. // Generate Huffman Code from Heap Tree.
2. // Input: Node Node, int code[], int top
3. // Output: Huffman Codes
4. Algorithm PrintCode(Node, code, top)
```

```
{
    if Node is a leaf do {
        for i := 0 to n do {
            print(code[i]);
        }
    }
    if (node->left) {
        code[top] = 0;
        printCodes(node->left, code, top + 1);
    }
    if (node->right) {
        code[top] = 1;
        printCodes(node->right, code, top + 1);
    }
}
```

c. Time Complexity:

The printing process prints n character's Huffman Code and traverse the Huffman tree. The time complexity would be **O(n)**.

d. Space Complexity:

The algorithm uses several integers, code array and a Heap tree. **The space complexity would be O(N).**

6. Overall Algorithm, Huff():

a. Abstract:

Gathering all elements in the above analysis, we can construct an algorithm to implement Huffman Code Encoding.

b. Algorithm:

```
1.  // Generate Huffman Code from Heap Tree.
2.  // Input: Node Node, int code[], int top
3.  // Output: Huffman Codes
4.  Algorithm Huff(data, freq, size)
5.  {
6.      for i := 0 to size do {
7.          HeapTree->array[i] := newNwode(data[i], freq[i])
8.      }
9.
10.     HeapTree = HeapSort(HeapTree, size);
11.     root = BMT(HeapTree);
12.     PrintCodes(root);
13. }
```

c. Time Complexity:

The time complexity for Huff() will be:

O(n) for initializing.

O(n lg n) for Heap Sorting.

O(n lg n) for setting Binary Merge Tree.

O(n) for printing Huffman Codes.

The total time complexity is **O(n lg n).**

d. Space Complexity:

The algorithm uses the spaces those algorithms I analyzed above takes. **The space complexity would be O(N).**

7. Time & Space:

	Huff()
Time complexity	**O(N lg N)**

Space complexity	**O(N)**