COP5536 Spring 2017

Report of Programming Project: Huffman Coding

March 30th, 2017

Name: You Zhou

UFID: 1448-0101

I. Introduction

In this project, I present a compression and decompression scheme based on Huffman coding to reduce the size of the data to be transferred from MyTube. I use three data structures: binary heap, 4-way cache optimized heap and pairing heap to implement the priority queue, and then compare the performance of them using the input data of *sample_input_large.txt.* I choose the data structure with the best performance to generate the frequency table and first programmed the encoder which takes *sample_input_large.txt* as input to be compressed and generate two files *encoded.bin* and *code_table.txt*. Then I programmed the decoder which takes the output of the encoder: *encoded.bin* and *code_table.txt* as input to generate the decoded tree, and using the decoded tree we can generate the decoded message from the encoded message. One file: *decoded.txt* will be generated by the decoder. The whole program is coded in Java without using any standard library container other than vector. I implemented all the three types of priority queue by myself, which will be described and explained in detail in this report.

II. Design

1. Overall design of the program

The first step is to generate Huffman codes. The algorithm is to build a Huffman tree that has the minimum WEPL (Weighted External Path Length). The weight is measured by the frequency of a word in the message to be encoded. In this way, the words with higher appearance frequency will be replaced by shorter codes, thus the size of the whole file will be reduced significantly (lossless data compression). To build the Huffman tree, a greedy algorithm will do the trick perfectly: Start with a collection of external nodes, each of which defines a different tree. Select two trees with the minimum weight and make them the children of a new node, whose weight is the sum of the weights of the children. Then add the new tree to the tree collection and repeat the process until only one tree left.

Then how to implement this algorithm?

We can use the data structure of priority queue, which is implemented by three kinds of heap structures: binary heap, 4-way cache optimized heap and pairing heap. First push all the nodes into the priority queue. Each time we push a node into the priority queue, a min-heapify is done to move up the node with smaller frequency. After pushing all the nodes into priority queue, poll two nodes with the minimum weight at one time and link them to a parent node and define the frequency of the parent node as the sum of the frequencies of the two children. Then we push back the parent node into the priority queue. Total number of trees will be reduced by 1. Repeat the process until only one tree is left, which is the final Huffman tree.

Then we need to generate the Huffman codes. We define the edge that connects the left child as 0 and the edge that connects the right child as 1. In order to build the code table, we just need to depth first search (DFS) the Huffman tree. After building the code table, we can encode the input file according to the code table. In order to decode the encoded file, we should build a decode tree according to the code table.

2. Binary heap

We can use the ArrayList structure to store the elements and construct the binary heap. I will explain the implementation of several key methods of binary heap, other methods are easy and trivial.

1) add():When we add an element, we can first set the index of the newly added element to the last index of the list. Then we can do the min-heapify to move the element to the proper index. The process of min-heapify is to compare the frequency of the element and its parent, if it is smaller, then swap the position of the two elements. The index of the parent of a node is $\frac{index-1}{2}$. The complexity of add() operation of ArrayList is O(1), and the complexity of min-heapify is $O(\log n)$. Total complexity for one element is $O(\log n)$.
2) poll(): For the *poll()* method, we get the root element and the last element. Then we store the root element and put the last element in the root, followed by moving down the root element until it satisfy the heap property. At last, return the stored root element. The index of the left child of a node is $2\times index+1$. Total complexity for poll one element is O(log n). O(1) for ArrayList.set(), which is used

for swap elements. And O(log n) for min-heapify.

3) compare(): When we compare two node, we only compare the frequency field of the nodes.

3. 4-way cache optimized heap

We also use the list structure to store the elements and construct the 4-way heap.

1) Initialization: In order to achieve the cache optimized heap, we should make sure that the siblings are in the same cache line, thus the number cache misses will reduce. So we can add three empty nodes when initializing the heap. Thus the index of the root starts from 3.
2) add(): The implementation of the add method of 4-way heap is basically the same as the binary heap. However, during the moving up, the index of the parent is $\frac{index+8}{4}$. Complexity is $O(\log n)$.
3) poll(): Basically the same as binary heap, however, when moving down, we need to compare between the four children and swap with the smallest child. The index of the leftmost child is $4 \times index - 8$. Complexity is O(log n).
4) clear(): After clearing the list we should add three empty node to keep the cache optimized.

4. Pairing heap:

For pairing heap, I use pointer to construct the heap.

1) CompareAndLink(): Compare the frequency of two nodes and make the node with larger frequency the leftmost child of the node with smaller frequency.
2) add(): Compare the frequency of the new node and the root node. Call the method *CompareAndLink()*. And the heap size is increased by 1. Complexity: O(1).
3) poll(): Get the root and remove it. Then combine the children of it in two pass scheme. This method will call *CompareAndLink()* several times. Complexity: O(n).
4) DoubleIfFull(): Use an array to store the siblings, initialize the size of the array by 5, and double the size if the array is full. Amortized complexity O(1).

III. Implementation

The project includes several classes:

1) Node.class:

```
public class Node {
    String ch;
    int freq;
    Node left, right;
    boolean isLeaf()
    }
```

I construct the node class to use basically as tree node, however, it contains the string field *ch* to store the word (number) to be compressed, and the integer field *freq* to store the frequency of a word and two Node fields *left* and *right* to point to the left child and right child. It contains a method *isLeaf()* to determine whether the node is a leaf node.

2) PairNode.class:

```
public class PairNode {
    String ch;
    int freq;
    PairNode leftChild, next, prev, left, right;
    }
```

This class is used to store the nodes of pair heap, the difference between it and theNode.class is that it has two more fields *next* and *prev* pointing to the siblings in double directions.

3) hNode.class:

```
public class hNode {
    hNode left;
    hNode right;
    String data;
    public void setData(String data)
    public String getData()
}
```

This class is used to build the decoded tree, which contains 3 fields and 2 methods. The hNode field *left* and *right* point to the left child and the right child. The String field *data* is used to store the value that only exists on the leaf node. The method *setData()* is used to set the value of a node. The method *getData()* is used to get the value of a node.

4) MinHeap.class:

```
public class MinHeap {
    List<Node> heap;
    public void add (Node o)
    public void offer (Node o)
    protected int stepUpHeap (int index)
    protected int compare (Node element, Node other)
    public void clear ()
    public boolean isEmpty ()
    public int size ()
    public Node peek ()
    public Node poll ()
    public void stepDownHeap (int index)
}
```

This class is used to build the data structure of binary heap (min heap). It calls into the Node.class, and implement basically all methods of priority queue. Bottom-up min-heapify and top-down min-heapify is implemented by *stepUpHeap()* and *stepDownHeap()*.

5) PairHeap.class:

```
public class PairHeap {
    private PairNode root;
    private PairNode [] treeArray = new PairNode[5];// store sibling nodes
    public boolean isEmpty ()
    public int size ()

    public void clear ()

    public void add (PairNode x)

    private PairNode compareAndLink (PairNode first, PairNode second) //make
tree with larger root leftmost child of the tree with smaller root

    private PairNode combineSiblings (PairNode firstSibling) //two pass meld

    private PairNode[] doubleIfFull (PairNode [] array, int index)

    public PairNode poll ()

}
```

This class is used to build the data structure of pairing heap (min pairing heap). It calls into the PairNode.class. The methods are explained by the annotations. *treeArray* is used to store the siblings. The size of it is initialized as 5, however, when the array is full, its size is doubled by calling doubleIfFull().

6) FourHeap.class:

```
public class FourHeap {

    List<Node> heap;

    public void add (Node o)

    public void offer (Node o)

    protected int stepUpHeap (int index)

    protected int compare (Node element, Node other)

    public void clear ()

    public boolean isEmpty ()

     public int size ()

     public Node peek ()

     public Node poll ()

     public void stepDownHeap (int index)

      public int getSmallestChild (int index)

 }
```

This class is used to construct the 4-way cache optimized heap. It calls into the Node.class, and contains basically the same methods as binary heap, however, the calculation of the index of parents and children is different. *getSmallestChild()* will return the index of the node with the smallest frequency.

7) HuffmanTree.class:

```
public class HuffmanTree {
   public hNode root;
   public void add (String data, String sequence)
   public List<String> getDecodedMessage (byte[] encoded)
}
```

This class is used to build the decoded huffman tree. It calls into the hNode.class. Add method is used to build the tree structure according to the Huffman code from the code table. *getDecodedMessage()* is used to decoded the binary file encoded.bin.

8) packEncodedChars.class:

```
public class packEncodedChars {
  byte[] bitstring;
  public static byte[] zhuan(String[] number, Map<String, String> codetable)
}
```

This class is used to transform the encoded string to binary file. The method zhuan takes the original number and codetable as input, and outputs binary file encoded.bin.

9) Encoder.class

```
public class Encoder {
   public static Map<String, String> compress(String[] number)
   public static void buildCode(Map<String, String> codetable, Node x, String s)
   public static void main(String args[])
}
```

This is the main class, the entry point of the whole project. For binary heap, it calls into Node.class, MinHeap.class, packEncoderChars.class. For 4-way cache optimized heap, it calls into Node.class, FourHeap.class and packEncoderChars.class. For pairing heap, it calls into PairNode.class, PairHeap.class and packEncoderChars.class. It takes *sample_input_large.txt* as input, *encoded.bin* and *code_table.txt* as output.

10) Decoder.class

```
public class Decoder {

    public static void main (String arg[])

}
```

This is another main class, another entry point of the project. It calls into the hNode.class and HuffmanTree.class. It takes *encoded.bin* and *code_table.txt* as input, *decoded.txt* as output.

IV. Testing

After running the Encoder, I successfully got the encoded.bin, whose size is 24.6 MB, and code_table.txt, whose size is 27.9 MB. After running the Decoder, I successfully got the decoded.txt, whose size is 69.6 MB. These sizes are exactly the same as the samples. The following table shows the performance of implementing the priority queue using three data structures. The unit of measurement is millisecond.

#Operations	Binary Heap	4-way Heap	Pairing Heap
1	1106	1092	1161
2	944	1327	1284
3	697	1042	819
4	693	1295	780
5	852	1028	828
6	703	1026	1306
7	766	1080	780
8	693	1035	1116
9	692	1044	926
10	733	1053	1012
Average	787	1102	1101

Assume there is n elements in the code table. As we analyzed before, the total complexity to build the Huffman tree using binary heap is O(n log n). The total complexity to build the Huffman tree using 4-way cache optimized heap is O(n log n). And the total complexity to build the Huffman tree using pairing heap is $O(n^2)$.

V. Conclusion and Analysis

The decoding algorithm I use is to build a Huffman tree using the code table. Define a root node, then read the Huffman code in the code table. If the digit is ‘0’, then define a new node and make it the left child of the previous node. If the digit is ‘1’, then define a new node and make it the right child of the previous node. The value of each Huffman code is stored in the leaf node. Let’s assume that there is m rows (codes) in the code table. Then the time and space complexity of building the decode tree is O(m). Then we use the decode tree to decode the encoded message. We start from the root node. If the binary digit is ‘0’, we go to the left child, if ‘1’, we go to the right child. When we reach the leaf node, we will be able to know the value, and restart from the root. In this way, the encoded message will be decoded. Let’s assume there is n values in the original file, and for each value to be decoded, we shall go from the root to the leaf. Because m << n, the total time complexity will be O(n log n).