Advanced Data Structures (COP 5536)

Spring 2017

Programming Project Report

Lakshmi Saranya Kalidindi

80946311

skalidindi@ufl.edu

Introduction

The project is to implement efficient encoder and decoder using a Huffman tree for any input files, be it large or small after evaluating which of the three priority queue structures gives best performance. The greedy approach is used for Huffman tree where the more frequently used words are placed at the top and the less frequently used ones are placed at the bottom of the tree.

This project is comprised of 3 files that has multiple classes in each file.

Compiling Instructions:-

1. Adsproject
   Compile = Javac Adsproject.java
   Run = java Adsproject
2. Encoder
   Compile = javac encoder.java
   Run = java encoder <inputFile>
3. Decoder
   Compile = javac decoder.java
   Run = java decoder <encoded.bin> <code_table.txt>

Function prototypes

1. Adsproject
   This file was created to test and analyze which priority queue structure is best in performance. The three structures, binary heap, 4 way cache optimized heap and pairing heap were implemented and is run 10 times to build a Huffman tree. This file is again run 5 times to get the average best time. I found out that four way heap with cache optimization came out to be the better.
2. Encoder
   Since I got four way heap to be the better performing data structure, Huffman tree is generated using this priority queue structure. The output of this file would be 2 files, encoded.bin(which contains the encoded message) and code_table.txt(which contains the code table used to generate the encoded message) and print statement that gives the time required to run the encoder on that input file.
   There are 4 classes in this file.

a) Mininode – This class has the data structure to hold the input initially.

The variables of this class are:-

- Int data – stores value of the decimal value from the input
- Long freq -stores the frequency or the number of times the decimal value is repeated

Member functions of this class are:-

- Constructor functions
- Int getKey() – It returns the variable data of that node.
- Long getFreq() – It returns the variable freq of that node.

b) HuffmanNode – This class has the data structure to hold the individual nodes of the huffman tree to be built.

The variables of this class are:-

- Int data – stores value of the decimal value from the input
- Long freq -stores the frequency or the number of times the decimal value is repeated

Member functions of this class are:-

- Constructor functions
- Int getData() – It returns the variable data of that node.
- Long getFreq() – It returns the variable freq of that node.

c) fourWayHeap – This class has all the functions that are required to build the Huffman tree and also to write to the output files.

Member functions of this class are:-

- Int getchild1() – It computes and returns the first child of that node.
- Int getchild2() – It computes and returns the second child of that node.
- Int getchild3() – It computes and returns the third child of that node.
- Int getchild4() – It computes and returns the fourth child of that node.
- Int getparent() – It computes and returns the parent of that node.
- minHeapifyFourWay(Arraylist<huffmannode> A, int i) – This function ensures that any insertion and deletion will retain the min heap property by checking from the given index, i.

- extractminFourway(Arraylist<huffmannode> A) – This function removes the root and calls minheapify and returns the node with minimum frequency.
- buildFourwayheap(Arraylist<huffmannode> A) – This method simply builds a four way heap given an array as input.
- buildHuffmanusingFourway(Arraylist<huffmannode> A) – This function creates the Huffman tree using the above methods.
- genCodetable(HuffmanNode n, int array[], int size, int lr) – This recursive method creates codewords traversing the tree and returns the code table to the main function.
- storeinHash(HuffmanNode n, int[] array, int size) – This method stores the data,codeword pair in a hashmap which is then given as output data.

d) Encoder – This class has the main function which first reads data from the input file, generates the frequency table and with the help of the above functions create a Huffman tree and code table and then prints them in their respective output files.

3. Decoder

   This file takes 2 files, encoded.bin and code_table.txt, as input and gives an output in a file named decoded.txt. The aim is to ensure that the file decoded.txt should be exactly same as the input given for the encoder. The first step is to create a decoder tree from the code table given as input. Then print the output file, converting the bytes from encoded.bin with the help of the decoder tree.

   There are 3 classes in this file.

a) Node - This class has the data structure to hold the individual nodes of the decoder tree to be built.

   The variables of this class are:-
   - Int data – stores value of the decimal value from the input
   - String codeword -stores the codeword for that particular data.
   - Node leftchild – stores the pointer to the left child.
   - Node rightchild – stores the pointer to the right child.

   Member functions of this class are:-
   - Constructor functions

- Int getData() – It returns the variable data of that node.
- String getCodeword() – It returns the codeword corresponding to that node.

b) decoderTree – This class implements 2 important functions that are required to decode the message from the input files.

Variables of this class are :-

- root – stores the root of the decoder tree.
- OTFILENAME – stores the filename of the output file to be created.

Member functions of this class are:-

- generateTree(Arraylist<huffmannode> A) – This function creates a decoder tree that is responsible to decode the message.
- decodeMessage(String INPUTFILE) – This function traverses the decoder tree and prints the output in the output file, decoded.txt.

c) decoder – This class has the main function which just reads the input files and using the above the classes and functions prints the output.

Performance analysis:

It was found out that the least running time to build a Huffman tree was taken by the four way cache optimized heap. Below is a detailed analysis of 3 of the runs(units – milliseconds):

	Four way heap	**Binary heap**	**Pairing heap**
Run 1	4738	5444	8728
Run 2	4235	4826	8140
Run 3	4840	4961	9002

Pairing heap takes the largest time since the tree can grow to any length. Also while extracting the minimum, all the children should be combined by a complex procedure called 2 pass scheme which supposedly gives better performance. Even though it is easier to implement, this queue structure did not show optimum performance. The four way heap takes lesser time than the other binary heap, though by a small difference. Even though it would seem like there would be more number of comparisons in four way heap, it would have to traverse a tree with smaller height in four way, (since $\log_4 n$ is smaller than $\log_2 n$). Also four way heap is cache optimized to get all the 4 children at a time.

The above table was obtained after taking the sample_input_large.txt file as the input. When run on storm.cise.ufl.edu, the encoder took 23 seconds and the decoder took roughly 27seconds. When an input file of 100 million lines is used as input, the encoder took 59 seconds whereas the decoder took around 235 seconds. These values were different when run on my own system. Even on thunder.cise.ufl.edu server, the values were increased.

Decoding algorithm:

There are 2 parts in this decoder algorithm.

1) Generate decoder tree

```
void generateTree(ArrayList<Node> A){
            for(int a=0;a<A.size();a++){                    //O(n)
                  Node r = this.root;
                  String code = A.get(a).getCodeword();
                  //traverse through the codeword
                  for(int i=0;i<code.length();i++){         //O(m)
                        if(code.charAt(i)=='1'){
                              if(r.rightchild==null)
                                    r.rightchild = new Node();
                              r=r.rightchild;
                        }
                        else {
                              if(r.leftchild==null)
                                    r.leftchild = new Node();
                              r=r.leftchild;
                        }
                  }
                  r.data = A.get(a).data;
            }
      }
```

The algorithm takes the code table as its input in the form of an arraylist and iterates through it. For each code word, each byte is checked if it is zero or one. Since according to the Huffman tree, if it has a left child then it can be accessed by 0 and the right child by 1. So if a 0 is witnessed, then we will check if there already exists a left child. If there is a left child, we simply traverse in that direction and move on to the next byte. Similarly if 1 is witnessed. This way we will end up in the position where the leaf is supposed to hold the data. Then we assign the data field of that node to the data of that code word.

2) Decode using the decoder tree and encoded.bin

Each line of the input consists of 8 bits of encoded data which are stored in a string. Now each bit is taken and likewise is traversed in the decoder tree. When met with a leaf, the code word that is stored at the leaf will be printed out to the output file and the node pointer is reset to the root since from the next bit, the traversal will start from the root again. Otherwise traversal occurs based on the bit, 0 denotes the left child and 1 denotes the right child.

Analysis:

1) There are 2 for loops that determine the complexity of this algorithm. The first for loop will take O(n) where n is the arraylist size and the second loop will take O(m) where m is the maximum possible code word. According to the handout, the input file can have up to 100,000,000 lines and each line will contain a decimal value in the range of 0 to 999,999. Hence n varies from 0 to 999,999 and m varies from 1 to 100,000,000. Therefore, the total complexity of this algorithm would be O(m+n) or the total number of bits in the file, code_table.txt
2) Similar to the above case, we can say that this algorithm goes through all the bits in the encoded.bin file since each bit will have to be evaluated and respective decoded message will be produced.
3) Other than the above, time would be consumed to read the input file in the main function to be sent to generate the decoder tree. The rest would be insignificant compared to the above mentioned time complexities when run on large input files.