Advanced Data Structures (COP 5536)
Spring 2017
Programming Project Report

Huffman Encoding & Decoding

Professor:
Dr Sartaj K Sahni

Submitted By:

Moulik Agarwal
UFID: 39826794
moulik90@ufl.edu

1. INTRODUCTION

The project aims to develop a data compression model to encode the data with an efficient scheme and then using a decoder to decode the encoded message back to the normal result. To implement such a scheme, we use Huffman Coding and Decoding algorithm to achieve our end goal. **Huffman coding** is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest **code** and the least frequent character gets the largest **code**. The algorithm runs in two phases, first phase involves encoding the given input while second phase runs the decoding algorithm to generate back the input.

2. ALGORITHM

2.1 ENCODING SCHEME & PROGRAM STRUCTURE

Huffman encoding works by building a binary tree of the input in such a way that the data resides in the leaf nodes and internal nodes determine the state. The binary created in the process is not an ordinary binary tree, it is a tree which generates prefix free codes for each data input. As provided by the instructor the could contain 10000000 lines and each line could contain a decimal integer ranging from 0 to 999,999. Each integer in the file is encoded by a Huffman code corresponding to the integer which is also a prefix free code. Below flow diagram depicts the working of an Encoder we have developed.

Following the above scheme the encoder generates two files as output i.e. encoded.bin and code_table.txt. The first file i.e. encoded.bin contains the input in an encoded format, while the code_table.txt contains unique integer values in the input with their corresponding encoding to be further used by the decoder to decode the encoded message. The encoded.bin isn't just a ordinary encoded file but has certain key features as well. The size of the encoded.bin is the lowest possible size to encode the input file as these are created based on a Huffman tree of least weight external path length.

2.1.1 PARSING INPUT & GENERATING FREQUENCY MAP

The first in building an encoder is to parse the input file and create a frequency map of the integers in the input file. Below is the function definition of the function used to parse the file and save frequency table in a HashMap.

```
-------------------------------------------------------------
public static HashMap<Integer, Integer> parseFile(String
fileName) throws IOException {...}
-------------------------------------------------------------
```

The function takes the file name as the input argument and return a HashMap of Integers having key as the Data value and Value as the frequency of that key. To read the file we use simple java FileReader and begin reading line one after another. As we read line we update the HashMap with the key and value respectively.

2.1.2 CREATE HUFFMAN TREE

As a next step in the encoding algorithm is to create a Huffman tree using the frequency table we have built so far. The algorithm begins with creating a binary tree of nodes. A node could either be a leaf node or an internal node. At the very beginning all the nodes are the leaf nodes with data value as each unique integer in the input file and their frequency in the input. During the process of algorithm, we require a priority queue to insert all the nodes and retrieve them each time with a *removeMin* operation. The general steps of the algorithm are as follows:

- Create a leaf node for each symbol and add it to the priority queue.
- While there is more than one node in the queue:
    - Remove the two nodes of lowest frequency from the queue
    - Create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequencies.
    - Add the new node to the queue.
- The remaining node is the root node and the tree is complete.

The complexity and the performance of tree generation depends on our choice of PriorityQueue used for creating the tree. In the project we evaluate which of the following priority queue provides the best performance: BinaryHeap, 4-way cache optimized heap and Paring Heap. 4-way cache optimized heap is a 4-Ary heap structure with each node having 4 children's but with a slight modification. 4-way cache heap initializes the array of nodes after leaving 2 slots empty. This way we avoid a cache miss each time we fetch children's of a node

since all the child's are fetched in single operation. The Node structure is shown below :

```
public static class Node implements Comparable{
          private final int digit;
          private final int frequency;
          private final Node left;
          private final Node right;

          Node(int digit, int frequency, Node left, Node right){
              this.digit = digit;
              this.frequency = frequency;
              this.left = left;
              this.right = right;
          }

          public boolean isLeaf(){
              return (left == null  &&  right == null);
          }
          @Override
          public int compareTo(Object o) {
             return this.frequency - ((Node) o).frequency;
          }
}
```

Each node has 2 integer data fields such as Integer value & its frequency and 2 child pointer i.e. right child pointer and left child pointer. Following are the results obtained using different priority queue structure during the Huffman tree creation:

Priority Queue	Time in milliseconds (ms) Input _small.txt	Time in milliseconds (ms) Input _large.txt
Binary Heap	~ 1	870
4-way Cache Optimized Heap	~ 1	621
Pairing Heap	~ 1	1350

Looking at the above results we conclude that Binary Heap and 4-way cache optimized heap outperform the pairing heap and comparatively Binary heap gives better runtime performance than the 4-way cache optimized heap.

The main functions created for implementing a Priority Queue are (Shown for Binary Heap , similar functions are implemented for other Priority Queue as well as submitted in the code):

- **insert() :** Adding element to the tree to build up a priority queue.It works in **O(log N)** where N is the number of elements in the tree. Below is the program structure for the same :

```
------------------------------------------------------------
public void insert (Key x) {...}
------------------------------------------------------------
```

  The key here refers to the type of value added to the priority queue data structure, for eg. In our implementation we do the insert as following :

```
prioQ.insert( new Node(c,freqs.get(c),null,null));
```

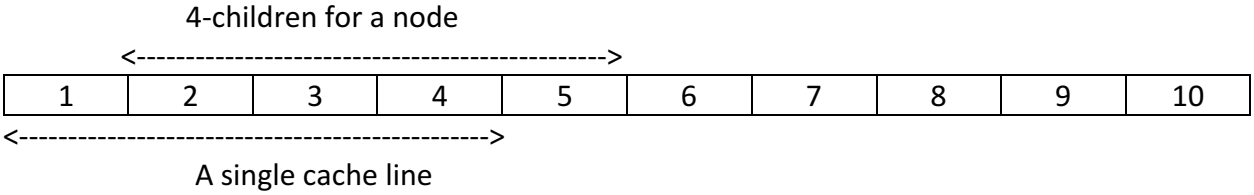
  Here key is the Node. The algorithm to insert into a Priority queue is as follows :
  - Add the element to the first missing element leaf
  - Switch it with its parent if its parent is larger and bubble the same operation up the tree until fixed
  - Repeat step 2

- **removeMin() / extract_min() :** The subroutine, finds the smallest element in the data structure and removes it from the tree and returns its value. The algorithm works in **O(log N)** time where N is the number of elements in the tree. Below is the structure of the function used in our implementation :

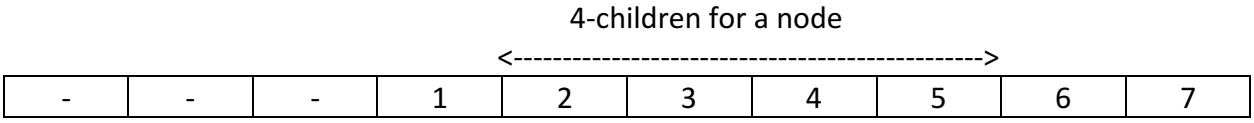
```
------------------------------------------------------------
public Key removeMin() {...}
------------------------------------------------------------
```

  The algorithm for remove the min element from the priority queue works as follows :
  - Remove the root element and save it to later return it
  - Remove the first last leaf in the tree. Place it at the root of the tree.
  - Now, move down the tree and restore the Heap property wherever not satisfied
  - Return the root value saved before

Considering a 4-way Cache optimized heap, we have similar operations with a slight modification. Consider an array of Node containing all the elements of the tree :



For the above scenario every parent has its parent in 2 cache lines therefore each time we fetch the child's of a node we incur a cache miss. To avoid this and increase the performance of the algorithm we shift the array by 2 slots so that every child of a node could be fetched in a single operation as shown below :



Following this approach we make the data structure work faster and improve the overall Priority Queue implementation.

Below is the function definition for the Huffman Tree creation :

```
private static Node buildHuffmanTree ( HashMap<Integer, Integer> freqs ) {...}
```

The *buildHuffmanTree* function takes HashMap containing key as integers and value as their corresponding frequencies, as the function argument. Using the above input HashMap we create the Huffman tree using the best performant priority queue. After creating the Huffman tree we then move to generating the Huffman codes for each input integer.

2.1.3 GENERATING HUFFMAN CODES

Using the above generated Huffman tree we now generate the Huffman codes corresponding to each input Integer in the input file. For generating codes we follow a simple scheme according to which we start from the root and follow the path to each of the leaf node. On the path to the leaf node, the left child of the current pointer is encoded as ”0” while the right child is encoded as “1”. On reaching the leaf node i.e. the data field, the code generated so far is the actual Huffman code for the data field. Once we have the Huffman code, we store it in a HashMap with key as the input data integer and value as the encoded code generated so far. The function structure of the above function is as follows :

```
private static void getHuffmanCode(Node node, String code, HashMap<Integer, String> encoding) {...}
```

The functions takes 3 values as the input arguments to it namely Root node of the Huffman tree , The code value created so far and the encoding HashMap contacting Huffman codes for each unique data (Integer). The function works recursively to generate Huffman codes corresponding to each leaf node in the tree. Once we are done with generating Huffman codes for each data integer we could move onto the last step of the execution to generate the output files.

2.1.4 GENERATING OUTPUT

After we are done creating the HashMap containing Huffman codes of each unique integer value in the input we begin writing out to encoded.bin file. To generate a better runtime performance in case of working on large input files we read input file line by line, get the encoding corresponding to it from the encoding hashmap, divide it into chunks to 8 bits and finally write it to the encoded.bin file. Below is the function prototype of the function generating encoded.bin file :

```
-------------------------------------------------------------
private static void generateEncodedBinFile(HashMap<Integer,
String> encoding, String fileName) throws IOException {...}
-------------------------------------------------------------
```

The function takes in 2 arguments, one is the encoding HashMap to lookup the encoding of the Integer and second is the file name of the output file.

The second part of the output file generation phase is to generate code_table.txt. Generating code_table.txt is straightforward given the encoding hashmap we have built earlier. Below is the function definition for the same:

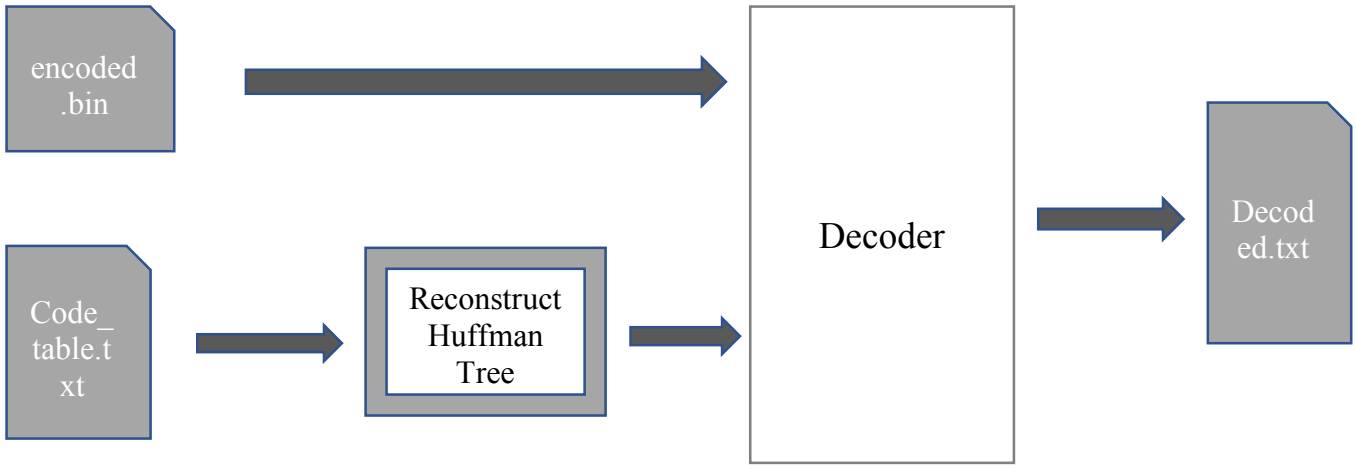
```
-------------------------------------------------------------
private static void generateCodeTable(HashMap<Integer, String>
encoding) throws IOException {...}
 -------------------------------------------------------------
```

The above functions takes one argument to it, which is an encoding HashMap containing Huffman code corresponding to each Integer value. To output the code_table.txt we iterate over the hashmap and retrieve all key value pairs, save them onto a string bulder and then once we are done iteration over the map we write the content created so far to the output file. Thus the overall algorithm runs in **O(N log N)** time where N is the number of items in the tree.

2.2 Decoding Scheme & Program Structure

This is a key step in the process of data compression to correctly decode the encoded message received from the encoder. Decoder is a fairly 2 step process where, first we need to reconstruct

the Huffman tree using the code_table received from the encoder and then decode the encoded message using this Huffman tree.

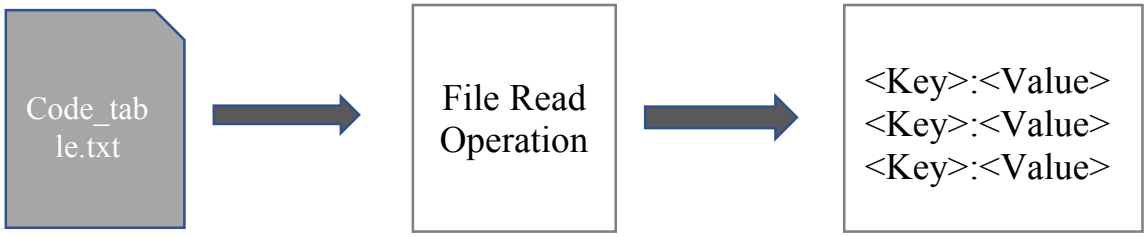


2.2.1 RECONSTRUCTING HUFFMAN TREE

This is a crucial step in decoding the message as this step determines the complexity and performance of our decoder. For reconstructing the Huffman tree from the code_table received from the encoder. The code_table.txt contains the Unique integers values and their corresponding Huffman code. Now, before we begin to reconstruct the tree, we first need to read the code_table.txt and convert it to a HashMap for our decoding purpose.

2.2.1.1 GATHERING HUFFMAN ENCODING

To begin reconstructing encoding hashmap, we read the code_table.txt file via file I/O operation in java and start saving values in a HashMap with key as the Integer and value as its corresponding encoding. The figure below depicts this operation more clearly:



Since we are now in the decoding phase, we might have to further process this HashMap for our purpose. We would be receiving the encoded message from the encoded.bin file and we need to decode this message with correct Integer values. To do this, we need to flip the key value pair so that we have a mapping of encoding to integer value i.e. encoding becomes the key and its corresponding Integer becomes the value.

Below is the function prototype of the same:

```
-----------------------------------------------------------------
private static void getCodeTable() throws IOException {...}
-----------------------------------------------------------------
```

Using the above function, we now have the HashMap to get the Integer corresponding to the given encoding.

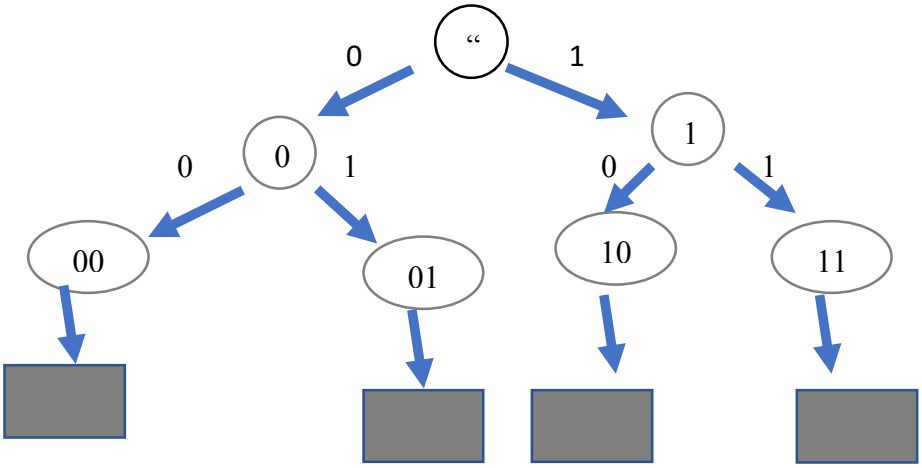
2.2.1.2 TRIE

To reconstruct the Huffman tree from the HashMap we would create a Trie data structure. Trie also called a prefix tree is a kind of search tree that is used to store dynamic sets and retrieve information quickly. Unlike any other search tree, each node in a Trie doesn't stores the key associated with that node however the position of the node defines the key it's is associated with. Below is the structure of Trie node used in our implementation:

```
class TrieNode {
    TrieNode[] arr;
    boolean isEnd;
    public TrieNode() {
        this.arr = new TrieNode[2];
    }

}
```

Since we would be working with only bits i.e. 0 or 1, we have an array of TrieNode which stores the information about the children pointer.



Below is the function prototype of function used to generate the Huffman tree :

```
-----------------------------------------------------------------
private static void createHuffmanTree () {...}
-----------------------------------------------------------------
```

In the above function we perform 2 tasks, one we create the code_map i.e.HashMap of values and integer and second is to insert each encoding into the Trie. The function prototype of the Trie insert is shown below:

```
public void insert (String word) {...}
```

The insert into a Trie takes one String parameter, which is the value to be inserted inside the data structure. The Insertion works on the principle of inserting prefixes in the data structure. For inserting any value, we perform a matching of bits with the existing structure. On receiving a "0" move left insert as a left child if not present otherwise move right and insert as a right child. On inserting the last bit, make the isEnd Boolean flag true so that we know it's a data leaf node. The runtime complexity of the Insert algorithm is **O(Key length)** however the memory requirement is **O(Key length*N)** where N is the number of keys in the Trie. This also becomes the overall complexity for the decoding Algorithm as well.

2.2.2 DECODER ALGORITHM

At the final step of decoding, we would use the Huffman tree created at the earlier step together with the encoded message received to generate the final output decoded text. Below is the function prototype of the same.

```
private static void runDecoder() throws IOException {...}
```

Inside the decoding algorithm, we read the encoded.bin file byte by byte to receive an 8-bit number. After converting the integer back to its string value we search the string in the Trie data structure. The function prototype of the same is shown below:

```
public int traverse (String s) {...}
```

Now, we do a lookup for the given string value in the Trie data structure built earlier. Inside the traverse function we iterate over the string and read bit one after the another. On receiving the bit as "0" move left i.e. access the left child of the parent, otherwise move right i.e. get the right child of the parent. At each step we check if the node is a leaf node, in case it is we return the index of the bit in the input string. We then extract the string using the index returned and do a lookup in the code_map HashMap built at the first stage of decoder to retrieve the Integer key associated with it and store it in the resultant string. Once we are done with reading the whole file ,we have the result in the appended string created so far. At the final step we just output the string as the decoded.txt.

3. ANALYSIS & FINAL RESULTS

Huffman encoding algorithm finds use in many day-to-day applications where we need to be able to compress the data using an efficient scheme and also be able to get the data out of the encoded message. The above project is built in Java with the help of inbuilt packages and libraries. The application runtime and complexity mainly depends the use of Priority Queue for creating Huffman tree at the encoder and the reconstruction of the Huffman tree at the decoder. To support the application for encoding large file in the order of few MB's we need to incorporate certain techniques so that the processor doesn't die because of operating on a huge data set.

To support large files certain level of optimization is performed at the time of file read/write I/O operation.

- First, create frequency HashMap at the time of reading the file line by line rather than first saving the input and then processing to generate the HashMap
- Second, inorder to encode the file , read the file again line by line and encode the Integer correspondingly using the encoding HashMap created previously.
- Writing the encoded value in byte chunks to generate an optimized file size.

The table below depicts the results in a simple manner at encoder and decoder using the input_large.txt .

	Priority Queue	Reading the Input file (ms)	Creating Huffman tree (ms)	Generating Huffman codes (ms)	Total Running Time (ms)
Encoder	Binary Heap	3539	653	559	13991
	4-way cache heap		717	523	14654
	Pairing Heap		1363	971	16499
Decoder		823	2741	-	45451

4. WORK ENVIRONMENT & COMPILATION INSTRUCTIONS

Hard Disk Space: 1 GB minimum or depends on the input file size
Memory: 512 MB
CPU: x86
OPERATING SYSTEM : Windows 10 , Mac OSX Sierra, Also tested on Linux
Complier : Javac

Running Instructions:

The program has been coded in JAVA and JVM compiler is used to compile the code and tested on storm.cise.ufl.edu, Windows 10 (Eclipse) and Mac OSX 10.12.3 platforms

To compile the program you can run the below command:

$ make

The above command will generate the class files for your program

To clear out the class files and the jar file you can run the below command

$ make clean

To execute the program run the following commands:

$ java encoder <input file name>

The above command would run the encoder.java on the provided input file and generate encoded.bin and code_table.txt file. Now to run the decoder run the following below command on the terminal :

$ java decoder encoded.bin code_table.txt

This would run the decoder and generate the decoded.txt file as the output.