COP 5536 ADVANCED DATA STRUCTURES

SPRING 2017

HUFFMAN ENCODING AND DECODING

**PRATHYUSHA KANUMALLA**

UFID: 33413100

**Problem Statement:**

To transfer huge amounts of data from a source to destination using variable length bits scheme instead of fixed length bits code scheme.

**Huffman Encoding:**

While Transferring data which can be repetitive, we use an optimizing scheme called Huffman encoding to encode enormous amount of data. It assigns variable length codes instead of fixed length codes so that transfer is effective. Hence we give a shorter code to more frequency words. Here the codes should be unique and also there should not be any overlap of codes within words. Hence we can effectively encode the data to bits and transfer bits stream. Thus the amount of data to be transferred will be drastically reduced. Thus the transferable cost would be the time to transfer variable length codes for each word and the mapping to each word.

**Algorithm for Encoding:**

The encoding part as mentioned should generate bits in a way as to represent high frequency words with lower number of bits. Thus we construct the tree in a way that lower frequency words are given higher number of words. Thus, we consider having a min heap and get the bits representation of words. Hence the encoding part will use heap to get the min elements first and construct a dummy node with frequency as the cost of constructing both the children. Thus we traverse after the tree construction to get the bits representation.

The following is the implementation of the encoder

public class encoder {

public encoder(String fileName) throws Exception {

To get the corresponding file and build a map of data and its frequency. Use this map and build a Huffman tree. This requires to get the minimum which can be achieved using a min heap implemented using pairing heap. After the Huffman tree is build , each data and its corresponding bit coding is saved into a file.

}

private void writeEncodeDataToFile(String fileName) throws Exception {

To write the encoded data into a encoded.bin file.

}

private void writeCodeTableToFile() throws Exception {

The code to bits mapping is written onto a file.

}

private void constructHuffManCodeMap(PairingEntry node, StringBuilder stringBuilder) { The Huffman tree is used to encode data to its corresponding bits.

}

public static void main(String[] args) throws Exception {

main method to calculate time for encoding and catch exceptions.

}

private static void getInput(HashMap<Integer, Integer> map, String fileName) throws Exception {

read input file and then populate a map of code and frequency.

}

**Huffman Decoding:**

The encoded data is transferred to the destination. Now the destination client should be able to decode the received binary file. The client requires the word to binary map to decode the binary file.

1.Then the decoding can be done by checking the map in O(1) time. But the hashing algorithm needs to consider the bits as the key and the algorithm should very effective to handle overlaps.

2. Instead we can use a tree with nodes in an order as to represent a 0 with left node and 1 with right node. Hence we form a tree and get the corresponding data from the tree.

This is similar to the construction of a digital trie data structure. Whenever a bit comes out we go the left child from root in case if the bits is 1, and to right child in case if the bit is 0. Thus the number of bits that needs to be travelling is the number of bits in the data. Thus the time taken is O(k). While encoding, we made sure that the high frequent words are given lower number of bits. The number of bit stream taken from the file is 8k and is mapped to corresponding data elements. As the data is recognized we write the data into the file.

**public** **class** decoder {

**public** **class** DecoderNode {

**int** data;

**public** DecoderNode leftChild, rightChild;

}

**public** decoder(String encoderFile, String cdeTableFile) **throws** Exception {

Generate map of code and its bits stream using the code table file. The encoded bin is read and bits are mapped to data. The data is written into a file.

}

**private** **void** decodeEncodedData() **throws** Exception {

method to decod.

}

**private** **void** writeBytesToDecoder() **throws** IOException {

method to write 8k bits into the file

}

**public** **static** **void** main(String[] args) **throws** Exception {

main method to calculate the times.

}

**private** **static** **void** getInput(Map<Integer, String> valueCodeMap, String fileName) **throws** Exception {

to get input.

}

}

**Analysis for the heap selection :**

Pairing heap took less time. Hence the encoder was implemented using pairing heap. The amortized costs are as follows.

|  |  |  |
| --- | --- | --- |
| Priority Queue Data Structure | insert(n) | removeMin() |
| Binary Heap | O(log2n) | O(log2n) |
| Four Way Cache Optimized Heap | O(log4n) | O(log4n) |
| Pairing Heap | O(1) | O(n) |

**Binary heap:**

Each node will have 2 children. The children of the nodes are implicitly greater than the node i.e., each child of the node is a min heap. The Binary Heap is effectively implemented in array.

**Four-way cache optimized heap:**

Instead of using a 2 degree heap which can take O(log2n), if we can increase the degree, so as to decrease the complexity. But, there will be a over head in number of hits to data. So, we optimize it by maintaining a four way cache. The cache size of L1 is 4. Hence we move the node to 3 bits so that the alignment of its children is perfect and the data hits required is decreased.

**Pairing Heap:**

 Although the asymptotic complexities of Pairing heap is worse when compared to the asymptotic complexities of Four way cached Priority Queue and Binary Heap, in practice the performance of Paring heap over a large series of inserts and deletes seemed to better than that of Binary heap and Four way cached heap. It is because of the Amortized time complexity of each operation of Paring heap. Every operation takes O(log n) amortized time.

|  |  |  |
| --- | --- | --- |
| Binary Heap | Four-cache Heap | Pairing heap |
| 1100 ms | 986 ms | 190 ms |

**Conclusion:**

In this project, the following are implemented.

1. Huffman tree construction : Binary Tree was constructed using pairing heap. We have evaluated the performance of 3 different priority queues which are Binary Heap, Four way cached heap and Pairing heap. Then the best one is used to build the Huffman encoder.

2.Encoder: Encodes data from the input file using bit codes generated from the Huffman tree.

3.Decoder: decodes data using data-bit code reference and encoded binary file.