Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

ADS Project Report

Introduction

We had to implement a message lossless compression system which consists of an encoder and decoder based on the Huffman Tree implementation. A Huffman Tree works with the help of a priority queue which allows it to build itself by identifying two consecutive minimum values and constructing the tree in a bottom up manner. The Priority Queue is implemented using three data structures: Binary Heap, Four Way Cache Optimized Heap and Pairing Heap. Each implementation performs differently in removing the minimum value. Using the data structure with the best implementation, we construct the Huffman Tree. Then, we read the input file, and use the Huffman tree to encode the input file into a compressed binary file and the code table file. The decoder uses the code table from the encoder to construct the binary trie, which is then used to decode the encoded file into the original input file.

Structure of the Program

1. **HuffmanNode**
   a. int data
   b. int frequency
   c. HuffmanNode leftChild, rightChild

2. **Binary Heap**
   a. Function Prototypes
      i. public void buildHeap(int[] frequencyArray)
      ii. private HuffmanNode removeMin()
      iii. private void heapify(int parent)
      iv. public void createHufffmanTree()
      v. private int peekChild(int index)
      vi. private int getParent(int index)
      vii. private int getChild(int i, int j)
      viii. public HuffmanNode peek()

3. **FourWayHeap**
   a. Function Prototypes
      i. public void buildHeap(int[] frequencyArray):
      This function takes a frequency array to build the four way heap. Every data point is inserted at the end of the heap, and is bubbled up to its correct position.

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

ii. private HuffmanNode removeMin():
This function removes the root of the heap, replaces it with the last element in the heap, and bubbles the new root down to its correct position in the heap. It returns the previous root as minimum element.

iii. private void heapify(int parent):
This function heapify's down the element positioned at the parent index. It compares the element at the parent with its children and swaps it with the smallest child, if required. It recursively calls itself with this smallest child.

iv. public void createHufffmanTree():
This function creates the Huffman tree using the four way heap. While the heap does not contain only one element i.e. the Huffman root, it continuously removes the minimum element twice, creates a new node with the summation of the frequencies of these two minimum elements, and reinserts the new node in the four way heap.

v. private int peekChild(int index):
This function returns the smallest child of the element at the specified index.

vi. private int getParent(int index):
This function returns the parent of the element at the specified index.

vii. int getChild(int i, int j):
This function gets the i'th child of the element at the j'th index.

viii. HuffmanNode peek():
This function returns the head of the four way heap.

4. PairingHeap
   a. Node
      i. HuffmanNode data
      ii. Node child
      iii. Node leftSibling, rightSibling
   b. Function Prototypes
      i. void insert(HuffmanNode data)
      ii. Node removeMin()
      iii. Node meldPairingHeap(Node p1, Node p2)
      iv. void buildHeap(int[] frequencyArray)
      v. void createHufffmanTree()

5. Encoder
   a. public void generateCode(HuffmanNode root, StringBuffer strBuf):
   This function is used to generate the codes from the Huffman tree for each individual data point. A hashmap maintains the specific data point and its corresponding code.

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

b. public void writeCodeTable(BuffereWriter codeFileWriter):
   This function is used to writer the data and its corresponding code to the **code_table.txt** file.

c. public void encodeInput(BufferedReader bufferedReader, BufferedOutputStream outputStream):
   This function is used to encode the input from the input file by looking up each data point's corresponding code, and writing it to the **encoded.bin** file.

d. public void testBinaryHeap(int[]frequencyTable):
   This function is used to test the performance of the binary heap.

e. public void testFourWayHeap(int[]frequencyTable):
   This function is used to test the performance of the four way heap.

f. public void testPairingHeap(int[] frequencyTable):
   This function is used to test the performance of the pairing heap.

6. Decoder
   a. DecoderNode
      i. String data
      ii. DecoderNode leftChild, rightChild
      iii. Boolean isLeaf
   b. Decoder
      i. private void codeTableToDecoderTree(BufferedReader codeTableReader):
         This function is used to read the **code_table.txt** file, extract each data point and its corresponding code and build the Trie.

      ii. private void insertIntoDecoderTree(String code, DecoderNode node, Integer data):
         This function is used to insert the data point and its code in its correct position in the Trie.

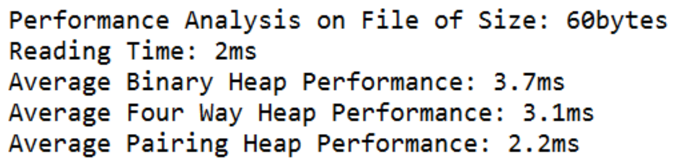
      iii. private void decodeData(BufferedWriter decodeFileWriter, FileInputStream encodedByteStream):
         This function is used to decode the data by reading the encoded.bin, traversing the Trie bit-by-bit and writing the data points into **decoded.txt.**

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

Performance Test Results

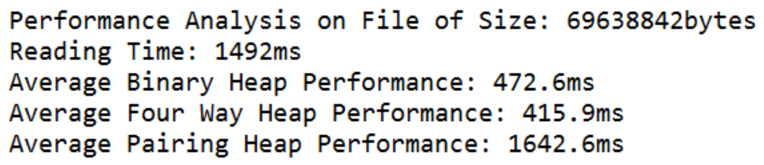
All of the three data structures: Binary Heap, Four Way Cache Optimized Heap, Pairing Heap were tested on files of three sizes: 60bytes, 66.4MB and 664MB. Each of them performed differently, and the results are as follows:

1. Small File Size(60 bytes)



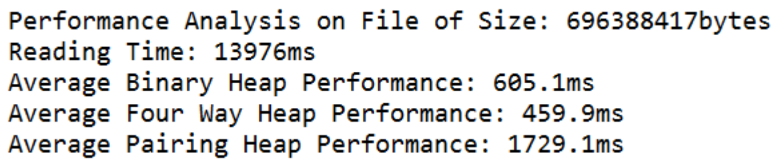
```
Performance Analysis on File of Size: 60bytes
Reading Time: 2ms
Average Binary Heap Performance: 3.7ms
Average Four Way Heap Performance: 3.1ms
Average Pairing Heap Performance: 2.2ms
```

2. Medium File Size(66.4MB)



```
Performance Analysis on File of Size: 69638842bytes
Reading Time: 1492ms
Average Binary Heap Performance: 472.6ms
Average Four Way Heap Performance: 415.9ms
Average Pairing Heap Performance: 1642.6ms
```

3. Large File Size(664MB)



```
Performance Analysis on File of Size: 696388417bytes
Reading Time: 13976ms
Average Binary Heap Performance: 605.1ms
Average Four Way Heap Performance: 459.9ms
Average Pairing Heap Performance: 1729.1ms
```

Explanation of the Results:

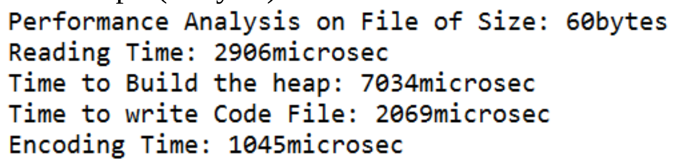
As shown above, pairing heap performs best on a small input. But as the input increases, pairing heap demonstrates the worst performance. The performance of all three data structures remains constant as the input increases from medium size to large size since the number of unique inputs will remain almost same, only causing their frequencies to increase. Pairing heap performs more number of movements on the data points due to its meld and remove min operations. D-ary heaps on the other hand, perform comparisons as well as swapping of the data points. On small input, pairing heap performs better since its easier to meld the root’s children everytime a removeMin is performed. Binary Heap performs worst since on every removeMin(), the newRoot would have to be bubbled all the way down to its correct position using heapify(). On the other hand, on large inputs, pairing heaps perform badly since the root will have many children, and melding each of them into a single root takes longer. Four way heap performs better than binary heap since on removeMin(), the newRoot will have to bubbled all the way down to its correct position using heapify(). However, the height of the heap will be $\log_4 n$ in comparison to $\log_2 n$ for Binary heaps.

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

Encoding

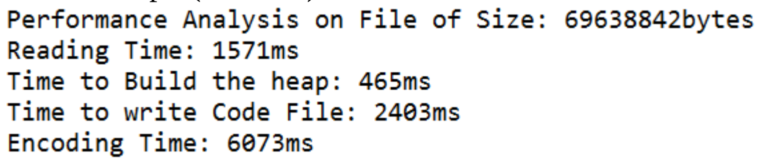
The following results demonstrates how long the encoding process takes on small(60bytes), medium(64.5MB) and large(664MB) inputs.

1. Small Input(60bytes)



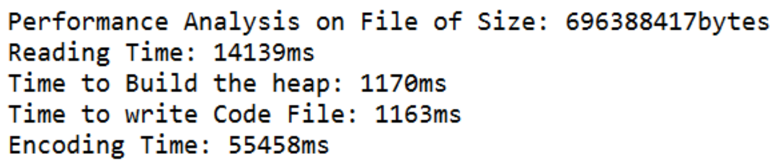
```
Performance Analysis on File of Size: 60bytes
Reading Time: 2906microsec
Time to Build the heap: 7034microsec
Time to write Code File: 2069microsec
Encoding Time: 1045microsec
```

2. Medium Input(64.5MB)



```
Performance Analysis on File of Size: 69638842bytes
Reading Time: 1571ms
Time to Build the heap: 465ms
Time to write Code File: 2403ms
Encoding Time: 6073ms
```

3. Large Input(664MB)



```
Performance Analysis on File of Size: 696388417bytes
Reading Time: 14139ms
Time to Build the heap: 1170ms
Time to write Code File: 1163ms
Encoding Time: 55458ms
```

Encoding uses a Four Way Cache Optimized heap to convert the input data into a Huffman tree. The algorithm for this procedure is explained below.

Algorithm:

1. Read input into a frequency table where index denotes the data point in the file and value at that index denotes its occurrences in the file.

2. Initialize a Four Way Heap
   a. Add dummy values at position 0,1,2. Set size = 3.

3. Build the four way heap by inserting each data point along with its frequency.
   a. Insert Algorithm for Four Way Heap
      i. Add the data point to the end of the four way heap.
      ii. HeapifyUp the newly inserted node.
   b. HeapifyUp Algorithm for Four Way Heap
      i. while(ParentNode > CurrentNode)

```
    Swap(ParentNode,CurrentNode)
    currentNode = ParentNode
```

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

4. Construct the Huffman Tree using the Four Way Heap
    a. Remove Minimum Element from Four Way Heap
        i. RemoveMin Algorithm for Four Way Heap
            - Extract root and replace with last element from the Four Way heap.
            - HeapifyDown from newRoot
        ii. HeapifyDown Algorithm for Four Way Heap
            - Compare ParentNode with ChildrenNodes
            - while(ChildrenNode[i] < ParentNode)
              ```
              Swap(ParentNode, ChildrenNode[i])
              ParentNode = ChildrenNode[i]
              ```

    b. Peek the Root from Four Way Heap
    c. Create newNode. Calculate frequency of newNode = minimumNode.frequency + root.frequency. Let newNode.LeftChild = minimumNode and newNode.Right Child = root. Replace root of heap with this newNode. HeapifyDown on newRoot.
    d. Repeat steps a – c until heap contains a single element. Return root as head of Huffman Tree.

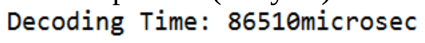
5. Generate Code Table
    a. Traverse the Huffman Tree to each individual leaf node. While traversing, on every left traversal, append bit 0 and right traversal append bit 1. At the leaf node, write the corresponding data point and code generated to the **code_table.txt** file.

6. Encode Input File
    a. Read the data point from the input.
    b. Lookup the data point in the code table and get the corresponding code.
    c. Write the code into the **encoded.bin** file.

<u>Decoding</u>

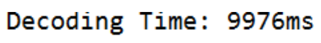
The following results demonstrates how long the encoding process takes on small(60bytes), medium(64.5MB) and large(664MB) inputs.

1. Small Input File(60bytes)



```
Decoding Time: 86510microsec
```

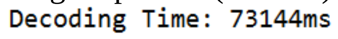
2. Medium Input File(64.5MB)



```
Decoding Time: 9976ms
```

Hussain Bootwala
UFID – 3925 9618
hbootwala@ufl.edu

3. Large Input File(664MB)



```
Decoding Time: 73144ms
```

Decoding uses a Trie to decode the encoded input file into decoded output file.

Algorithm:

1. Initialize an empty Trie.
2. Read the code table file and build the Trie.
   a. Read a line from the code table and extract the data point and its corresponding code.
   b. Insert the data point into the Trie.
   c. Algorithm for inserting in a Trie:
      i. Start temp = root.
      ii. If i'th bit = 0,
         (1) if leftChild of temp does not exist,
             Create a new branchNode if leftChild does not exist, and leftChild = branchNode.
         (2) temp = leftChild
      iii. If i'th bit = 1,
         (1) if rightChild of temp does not exist,
             Create a new branchNode if rightChild does not exist, and rightChild = branchNode.
         (2) temp = rightChild
      iv. Once all bits are traversed, create a new leaf node with the data point. If the last bit was 0, attach newNode to leftChild of temp, else attach to the rightChild of temp.
   d. Repeat the above procedure for all the data points in the code table.
3. Decode the encoded file into the output file.
   a. Read all bytes of the encoded file.
   b. Set temp = root.
   c. If i'th bit is 0, set temp to its left child. Else, if i'th bit is 1, set temp to its right child.
   d. If temp is a leaf node, write the corresponding data point in the leaf node to the decoded file.
   e. Iterate over each bit in the byteArray.

Complexity:

- The Trie has a complexity of O(MN) for construction, where M denotes the maximum number of bits required to encode each data point, and N denotes the number of unique data points in the input file.
- The Trie has a complexity of O(M logN) to extract a data point using its corresponding code.

Conclusion

I have successfully implemented the Huffman Tree Lossless Compression Algorithm with the help of a Four Way heap. After encoding, the input data gets compressed to almost half of its size.