Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

ADS Project Report

Introduction

We used Huffman Coding to implement a lossless message compression system consisting of an encoder and decoder. A Huffman Tree takes the help of a priority queue. This allows the tree to build itself by extracting two consecutive minimum values. The tree gets constructed in a bottom up manner.
We implement the Priority Queue using three data structures: Binary Heap, Four Way Cache Optimized Heap and Pairing Heap. Each implementation performs differently in removing the minimum value. We then select the data structure with the best performance and construct the Huffman Tree using the same data structure. We read the input file and use the constructed Huffman tree to encode the input file into a compressed binary file and the code table consisting of the respective codes. The code table is used by the decoder to reconstruct the original input file using a binary trie.

Machine Details

1. Processor
2.20 GHz core Intel Core i7 processor
64-bit OS
8GB RAM

2. Operating System: Windows 10

3. Compiler Used
Java Compiler version : 1.8.0_111
Java Runtime Environment Version : 1.8.0_111

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

Structure of the Program

1. HuffmanNode
    - int data
    - int frequency
    - HuffmanNode left, right

2. Binary Heap
    - Function Prototypes
        - void buildHeap(int[] freqArr)
        - HuffmanNode removeMin()
        - void minHeapify(index)
        - void generateHufffmanTree()
        - int getChild(int index)
        - int parent(int index)
        - int child(i, j)
        - HuffmanNode peep()
        - Void setRoot(HuffmanNode smallNode)

3. FourWayHeap
    - Function Prototypes
        - void buildFourWayHeap(int[] frequency): The input is a frequency array which has the frequencies of each data element. Each data element is inserted at the end of the heap, and then bubbled up to its correct position.

        - HuffmanNode removeMin(): The root of the heap is extracted and replaced with the last element in the heap. The new root is then bubbled down to its correct position in the heap. The previous root is returned.

        - void minHeapify(int index): The element position is heapified at the parent index. The element at the parent is compared with its children and then carries out a swap if required. This function is recursive and repeatedly calls itself with this best child.

        - void generateHufffmanTree(): This function generates the Huffman tree using the four way heap. It continuously removes the minimum element twice, creates a new node with frequency equal to the sum of the frequencies of these two minimum elements, and reinserts the new node in the four way heap.

        - int parent(int index): This function returns the parent of the element at the given index.

        - int child(int i, int j): This function returns the child of the element at the given index.
        - int getChild(int index): This function returns the smallest child of the element at the given index.

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

    - HuffmanNode peek(): This function returns the head of the heap.

    - Void setRoot(HuffmanNode minimumNode): This function sets the root of the heap to the minimumNode and then bubbles the minimumNode all the way down to its correct position.

4. PairingHeap
    - Node
        - HuffmanNode data
        - Node child
        - Node left, right
    - Function Prototypes
        - void insert(HuffmanNode data)
        - Node removeMin()
        - Node meld (Node p1, Node p2)
        - void constructHeap(int[] f)
        - void generateHufffmanTree()
        - void generateCode(HuffmanNode head, String code)
        - void display()

5. Encoder
    - void generateCode(HuffmanNode root, StringBuffer sb): This function uses a hashmap to store each data element and its corresponding code. This is used to generate the codes from the Huffman tree for each individual data element.

    - void writeCodeTable(BufferedWriter fileWriter): This function is used to write the data element and its corresponding code to **code_table.txt** file.
    - void encodeIn(BufferedReader reader, BufferedOutputStream filewriter): This function encodes the input by looking up each data point's corresponding code, and writing it to the **encoded.bin** file.
    - void testBinaryHeap(int[]frequencies): This function is used to test the performance of the binary heap.
    - void testFourWayHeap(int[]frequencies): This function is used to test the performance of the four way heap.
    - void testPairingHeap(int[] frequencies): This function is used to test the performance of the pairing heap.

6. Decoder
    - DecoderNode
        - int data
        - DecoderNode leftChild, rightChild
        - Boolean isEnd

    - DecoderTree

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

    - Void insert(String code, DecoderNode node, int data): This function is used to insert the data element and the corresponding code in the correct position in the Trie.
- Decoder
    - Void decoderTree(BufferedReader reader, DecoderTree tree): This function constructs the decoder tree.

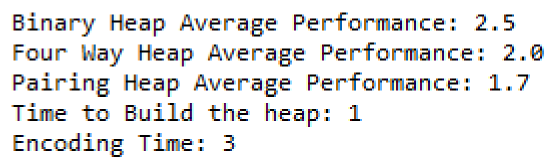
    - Void writeDecodedData(BufferedWriter fileWriter, FileInputStream encoded, DecoderTree tree, int size): This function is used to decode the data by reading the encoded.bin, traversing bit-by-bit and writing the data elements into **decoded.txt.**

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

Performance Test Results

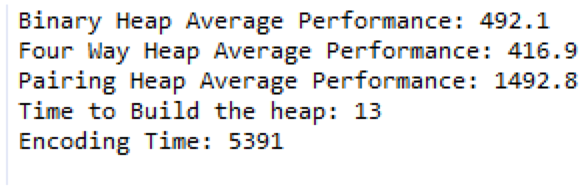
All three data structures Binary Heap, Four Way Cache Optimized Heap, Pairing Heap were tested on files of three different sizes: 60bytes, 66.4MB and 664MB.
Their performance was analyzed and the results are :

1. Small File Size(60 bytes)



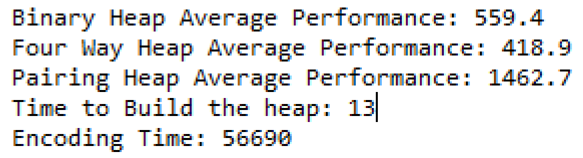
```
Binary Heap Average Performance: 2.5
Four Way Heap Average Performance: 2.0
Pairing Heap Average Performance: 1.7
Time to Build the heap: 1
Encoding Time: 3
```

2. Medium file size(66.4 MB)



```
Binary Heap Average Performance: 492.1
Four Way Heap Average Performance: 416.9
Pairing Heap Average Performance: 1492.8
Time to Build the heap: 13
Encoding Time: 5391
```

3. Large file size (664 MB)



```
Binary Heap Average Performance: 559.4
Four Way Heap Average Performance: 418.9
Pairing Heap Average Performance: 1462.7
Time to Build the heap: 13
Encoding Time: 56690
```

As demonstrated above, pairing heap has the best performance when the input file size is small. However, its performance deteriorates as the input file size increases. As the input increases from medium size to large size, the performance of all three data structures remains almost the same. This is because the number of unique inputs will remain the same, only causing their frequencies to increase.
Therefore, we proceed with the Four Way Cache Optimized Heap to construct the encoder and decoder.

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

Encoding

We use a Four Way Cache Optimized heap to convert the input data into a Huffman tree.
The algorithm is given below:

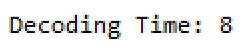
1. Read the input and insert it into a frequency table. Index stands for the data element in the file and value gives the number of times it occurs in the file.
2. Initialize the heap
   i. Add dummy values at index 0,1,2. Set the index of the root at 3.
3. Build the heap by inserting each data element and its corresponding frequency.
   i. Insert Algorithm:
      a. Add the data element to the end of the heap.
      b. HeapifyUp from the index of the newly inserted node.
   ii. HeapifyUp Algorithm:
      a. Compare the current node with its respective parent node
      b. while(parentNode > currentNode)
         Swap(parentNode,currentNode)
         currentNode = parentNode
4. Construct the Huffman Tree
   i. Remove Minimum Element
      a. RemoveMin Algorithm:
         - Extract the root node
         - Replace it with the last element from the heap.
         - HeapifyDown from the new root
      b. HeapifyDown Algorithm for Four Way Heap
         - Compare ParentNode with ChildrenNodes
         - while(ChildrenNode[i] < ParentNode)
           Swap(ParentNode, ChildrenNode[i])
   ii. Peep the root node
   iii. Create a new node. Calculate the frequency of the new Node as the sum of the minimum frequency and frequency of the root. Let the left child of the new node be minimum node and the right child be the root. Replace root with the new node followed by HeapifyDown on the new root.
   iv. Repeat steps a – c until the heap contains a single element. Return the root as the new head.
5. Generate Code Table
   i. Traverse the constructed Huffman Tree to each leaf node. On traversing left, append bit 0 and on traversing right, append bit 1.
   ii. At the leaf node, write the corresponding data element and the generated code to the code_table.txt file.
6. Encode Input File
   i. Read the data element from the input.
   ii. Lookup the data element in the code table to get the corresponding code.
   iii. Write the code into the encoded.bin file.

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

Decoding

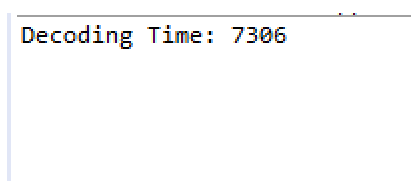
The following results demonstrates how long the encoding process takes on small(60bytes), medium(64.5MB) and large(664MB) inputs.
The following outputs are given in Milliseconds.

1. Small Input File(60bytes)



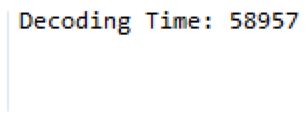
```
Decoding Time: 8
```

2. Medium Input File(64.5MB)



```
Decoding Time: 7306
```

3. Large Input File(664MB)



```
Decoding Time: 58957
```

We use a Trie to decode the encoded input file.

Algorithm:

1. Initialize an empty Trie.
2. Read the code_table file and construct the Trie.
   a. Read a line from the code table and extract the data element and the corresponding code.
   b. Insert the data element into the Trie.
   c. Algorithm for inserting in the Trie:
      i. temp = root.
      ii. If ith bit = 0:
         a) check if the right child of temp exists.
         b) Create a new branch node if the left child does not exist, and left = branchNode.
         c) temp = left
      iii. If ith bit = 1:
         a) check if the right child of temp exists.
         b) Create a new branch node if the left child does not exist, and left = branchNode.
         c) temp = right
      iv. After traversing all the bits, create a new leaf node with the data element. If the last bit was 0, attach newNode to left of temp, else it gets attached to the right of temp.
   d. Repeat the above steps until all the data elements in code_table are covered.

Subhrima Bhadury
UFID: 0698-8273
subhrima411@ufl.edu

3. Decode the encoded file:
    a. Read all bytes of the encoded file.
    b. Set temp = root.
    c. If ith bit is 0, set temp to its left child. Else, if ith bit is 1, set temp to its right child.
    d. If temp is a leaf node, write the data element in the leaf node to the output file.
    e. Continue for each bit in the byte array.

Complexity:

- The Trie has a complexity of O(M*N) to construct where M stands for the maximum number of bits required to encode each data element, and N stands for the number of unique data elements in the input file.
- The Trie has a complexity of O(M logN) to extract the data element using its corresponding code.

Thus, 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.