Algorithm Analysis

Wesley Anastasi

December 3, 2023

Description

The algorithm implements Huffman coding for compression and decompression of files.

Data Structure

The algorithm utilizes a priority queue to construct a Huffman tree. Additionally, it uses various data structures such as maps to store Huffman codes.

Algorithm

```
// Node structure for Huffman tree
struct Node {
  int count
  int data
  Node left
  Node right
  Node(count, data)
}

// Comparator for priority queue
struct CompareNodes {
  bool operator()(Node left, Node
right)
}

// Function to generate Huffman codes
generateHuffmanCodes(Node root,
Map huffmanCodes, String
currentCode)
  if root is null
    return
  if root has no left and right children
    huffmanCodes[root.data] =
currentCode
  generateHuffmanCodes(root.left,
huffmanCodes, currentCode + "0")
  generateHuffmanCodes(root.right,
```

```
huffmanCodes, currentCode + "1")

// Function to write Huffman tree to
output file
writeTree(Node node, OutputFile
outputFile)
   if node is null
      outputFile.put(0)
      return
   else
      outputFile.put(1)
      outputFile.put(node.data)
      writeTree(node.left, outputFile)
      writeTree(node.right, outputFile)

// Function to write encoded data to
output file
writeData(OutputFile outputFile,
InputFile inputFile, Map
huffmanCodes)
   String code
   while inputFile has more bytes
      byte = inputFile.get()
      code += huffmanCodes[byte]

   int size = code.size()
   outputFile.write(size)

   for i = 0 to code.size() step 8
      bits = convert code[i to i+7] to 8
bits
      encodedByte = convert bits to
byte
      outputFile.put(encodedByte)

// Function to read Huffman tree from
input file
readTree(InputFile inputFile) returns
Node
   marker = inputFile.get()
   if marker is 0
      return null
   else
      symbol = inputFile.get()
      node = new Node(0, symbol)
      node.left = readTree(inputFile)
```

```
      node.right = readTree(inputFile)
      return node

// Function to perform Huffman
compression
huff(source, destination)
   chunkCounts = array of zeros
   inputFile = open source
   while inputFile has more bytes
      byte = inputFile.get()
      chunkCounts[byte]++

   minHeap = priority_queue of Nodes
using CompareNodes
   for i = 0 to chunkCounts.size() - 1
      if chunkCounts[i] > 0
         node = new
Node(chunkCounts[i], i)
         minHeap.push(node)

   while minHeap.size() > 1
      left = minHeap.top()
      minHeap.pop()
      right = minHeap.top()
      minHeap.pop()

      mergedNode = new
Node(left.count + right.count, -1)
      mergedNode.left = left
      mergedNode.right = right
      minHeap.push(mergedNode)

   huffmanCodes = empty map

generateHuffmanCodes(minHeap.top()
, huffmanCodes)

   outputFile = open destination
   outputFile.put(5) // Magic Number
   writeTree(minHeap.top(),
outputFile)
   writeData(outputFile, inputFile,
huffmanCodes)
   close inputFile
   close outputFile
```

```
// Function to perform Huffman
decompression
unhuff(source, destination)
   inputFile = open source
   magicNumber = inputFile.get()
   if magicNumber is not 5
      print "Invalid file"
      return

   root = readTree(inputFile)
   huffmanCodes = empty map
   generateHuffmanCodes(root,
huffmanCodes)

   outputFile = open destination
   size = read size from inputFile
   bits = read encoded bits from
inputFile
   bits = bits.substr(0, size)

   current = root
   for bit in bits
      if bit is '0'
         current = current.left
      else
         current = current.right

      if current has no left and right
children
         outputFile.put(current.data)
         current = root

   close inputFile
 close outputFile
```

Analysis

Input N	Size of the input file
Basic Operation	Reading and processing each byte of the input file
Summation or Recurrence Relation	O(N)

Worst Case Analysis

The worst-case time complexity is dominated by the construction of the Huffman tree. It is O(K log K), where K is the number of unique bytes in the file. The space complexity is also O(K log K) due to the data structures used.

Best Case Analysis

The best-case time complexity is O(N), where N is the size of the input file. This occurs when the input file is small, and the overhead of tree construction becomes negligible. The space complexity remains O(K log K).