Algoritmo Voraz: Codificación de Huffman

Sebastián López Montenegro (97500)

14 de Noviembre 2024

1 Problema y contexto del problema

La *codificación de Huffman* es un algoritmo que se utiliza para asignar códigos binarios a los caracteres de un conjunto de datos de manera eficiente. Su objetivo principal es reducir el tamaño de los datos al asignar códigos más cortos a los caracteres más frecuentes y códigos más largos a los menos frecuentes.

Esto resuelve el problema de la codificación eficiente mediante la construcción de un árbol binario, llamado *árbol de Huffman*, donde cada carácter se representa en las hojas del árbol. El algoritmo es muy eficiente y se aplica principalmente en compresión de texto, imágenes y otros tipos de datos donde se pueden identificar patrones de frecuencia en los símbolos.

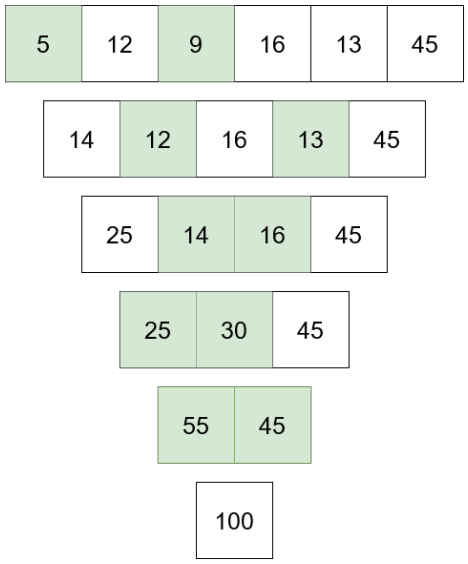
Problema: Dado un conjunto de caracteres con sus respectivas frecuencias de aparición, se debe asignar a cada carácter un código binario, de manera que el mensaje completo quede representado con el menor número posible de bits.

2 Explicación gráfica

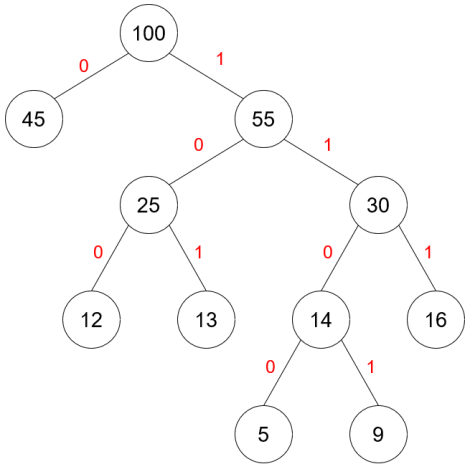
El proceso de codificación de Huffman se puede ilustrar mediante un árbol binario donde las frecuencias de los caracteres guían la construcción del árbol. El algoritmo sigue estos pasos:

1. Inicialmente, cada carácter se trata como un nodo con su frecuencia.
2. Se seleccionan los dos nodos con las frecuencias más bajas y se combinan en un nuevo nodo cuyo valor es la suma de las frecuencias de los nodos combinados.
3. Este nuevo nodo se inserta de nuevo en el conjunto de nodos y el proceso se repite hasta que solo quede un nodo, que será la raíz del árbol de Huffman.
4. A cada nodo se le asigna un código binario basado en su posición en el árbol: un ‘0‘ para el subárbol izquierdo y un ‘1‘ para el subárbol derecho.

A continuación, se muestra un ejemplo gráfico de la construcción del árbol de Huffman paso a paso y el árbol final:



Se puede observar que se escoge en la primera tabla las dos frecuencias menores (Coloreadas de color verde en la imagen) para luego sumarlas, así sucesivamente se va creando una nueva tabla que repite el proceso hasta llegar a un solo elemento.



Luego se construye el árbol respecto a las tablas anteriores. Entonces los caracteres más frecuentes tienen códigos más cortos, mientras que los menos frecuentes tienen códigos más largos, lo que optimiza la representación binaria del mensaje.

3 Implementación en Python

```python
class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

class HuffmanCoding:
    def __init__(self, frequencies):
        self.frequencies = frequencies
        self.codes = {}
        self.root = None

    def build_tree(self):
        # Create a node list
        nodes = [Node(char, freq) for char, freq in self.frequencies.items()]

        # Make huffamnn tree
        while len(nodes) > 1:
            # Sort nodes by frequency
            nodes.sort(key=lambda x: x.freq)

            # Take both nodes with lower frequency
            left = nodes.pop(0)
            right = nodes.pop(0)

            # create a combined node
            merged = Node(None, left.freq + right.freq)
            merged.left = left
            merged.right = right

            # add new node to list
            nodes.append(merged)

        # last node is the tree´s root
        self.root = nodes[0]

    def generate_codes(self, node=None, current_code=""):
        if node is None:
            node = self.root

        if node.char is not None:
            self.codes[node.char] = current_code
            return

        # Travel across the left and right subtrees
        self.generate_codes(node.left, current_code + "0")
        self.generate_codes(node.right, current_code + "1")

    def get_codes(self):
        if not self.codes:
            self.generate_codes()
        return self.codes

if __name__ == "__main__":
    frequencies = {
        'a': 5,
        'b': 9,
        'c': 12,
        'd': 13,
        'e': 16,
        'f': 45
    }

    huffman = HuffmanCoding(frequencies)
    huffman.build_tree()
    codes = huffman.get_codes()

    print("Códigos de Huffman:")
    for char, code in codes.items():
        print(f"{char}: {code}")
```

Tenemos entonces en la implementación una clase Node, que representa un

nodo del árbol de Huffman. Cada nodo tiene tres atributos: el carácter asociado (char), su frecuencia (freq) y punteros a sus subárboles izquierdo (left) y derecho (right). Luego, la clase HuffmanCoding encapsula el proceso de construcción del árbol y la generación de códigos. El constructor de esta clase inicializa un diccionario de frecuencias (frequencies), un diccionario vacío para almacenar los códigos de Huffman (codes), y un atributo para la raíz del árbol (root).

El método build_tree construye el árbol de Huffman, como se observó en imágenes anteriores. Crea una lista de nodos a partir del diccionario de frecuencias, donde cada nodo representa un carácter y su frecuencia. Luego, mientras haya más de un nodo, se ordenan los nodos por frecuencia, se extraen los dos nodos de menor frecuencia y se combinan en un nuevo nodo donde la frecuencia es la suma de las frecuencias de los nodos hijos. Este proceso continúa hasta que queda un único nodo, que se convierte en la raíz del árbol.

El método generate_codes genera los códigos binarios para cada carácter. Comienza desde la raíz del árbol y recorre recursivamente los subárboles izquierdo y derecho. Si encuentra una hoja le asigna al carácter el código binario acumulado hasta ese punto, usando 0 para moverse hacia la izquierda y 1 para la derecha. Los códigos generados se almacenan en el diccionario codes.

4 Resultados y conclusiones

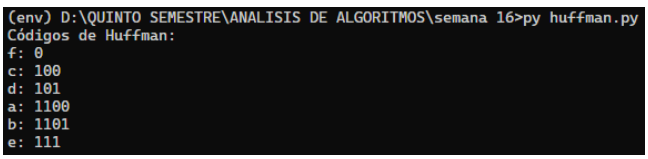
La implementación del algoritmo de Huffman produce una tabla de códigos binarios optimizados para cada carácter basado en sus frecuencias. Por ejemplo, para las frecuencias de entrada del conjunto de caracteres:

```
{'a': 5, 'b': 9, 'c': 12, 'd': 13, 'e': 16, 'f': 45}
```

La salida de los códigos de Huffman podría ser:

```
Códigos de Huffman:
a: 1100
b: 1101
c: 100
d: 101
e: 111
f: 0
```

y estos códigos los verificamos con los obtenidos de la implementación y se confirma que ha quedado bien:



Esto muestra cómo los caracteres más frecuentes tienen códigos más cortos, lo que reduce significativamente el tamaño total de los datos. En el caso de un mensaje con una distribución de frecuencias desequilibrada, este tipo de codificación puede ahorrar una cantidad considerable de espacio, mejorando la eficiencia en términos de almacenamiento y transmisión.

A continuación se muestra cómo el peso en bits disminuye significativamente:

La codificación se compara con la representación ASCII, donde cada carácter ocupa 8 bits independientemente de su frecuencia. Para calcular el peso total en bits usando ASCII, multiplicamos la frecuencia de cada carácter por 8 y sumamos los resultados:

$$\text{Bits totales (ASCII)} = 5 \cdot 8 + 9 \cdot 8 + 12 \cdot 8 + 13 \cdot 8 + 16 \cdot 8 + 45 \cdot 8$$

$$\text{Bits totales (ASCII)} = 40 + 72 + 96 + 104 + 128 + 360 = 800\,\text{bits}.$$

En la codificación de Huffman, la longitud de los códigos asignados a cada carácter depende de su frecuencia. Según los códigos del ejemplo:

- `a`: longitud 4 (`1100`), frecuencia 5,
- `b`: longitud 4 (`1101`), frecuencia 9,
- `c`: longitud 3 (`100`), frecuencia 12,
- `d`: longitud 3 (`101`), frecuencia 13,
- `e`: longitud 3 (`111`), frecuencia 16,
- `f`: longitud 1 (`0`), frecuencia 45.

El peso total en Huffman se calcula multiplicando la frecuencia de cada carácter por la longitud de su código y sumando los resultados:

$$\text{Bits totales (Huffman)} = 5 \cdot 4 + 9 \cdot 4 + 12 \cdot 3 + 13 \cdot 3 + 16 \cdot 3 + 45 \cdot 1$$

$$\text{Bits totales (Huffman)} = 20 + 36 + 36 + 39 + 48 + 45 = 208\,\text{bits}.$$

El ahorro total en bits es la diferencia entre el peso en ASCII y el peso en Huffman:

$$\text{Ahorro} = 800 - 208 = 592\,\text{bits}.$$

Esto representa un ahorro porcentual de:

$$\text{Porcentaje de ahorro} = \frac{592}{800} \cdot 100 \approx 74\%.$$

Este ahorro se da porque la codificación de Huffman asigna códigos mucho más cortos a los caracteres más frecuentes y así optimiza la representación de los datos. En cambio, la codificación ASCII utiliza una longitud fija de 8 bits por carácter, lo que es menos eficiente para distribuciones de frecuencia.