

Códigos de Huffman: Algoritmo Voráz

Juan Pablo Silvestre - 99127

16/11/2024

1 Resumen

La *codificación de Huffman* es un algoritmo de compresión sin pérdida de datos, como una cadena de caracteres. La idea es asignar códigos de longitud variable a los caracteres de entrada; las longitudes de los códigos asignados se basan en las frecuencias de los caracteres correspondientes. La construcción de este algoritmo se da mediante un árbol binario donde cada carácter representa las hojas del árbol.

2 Problema

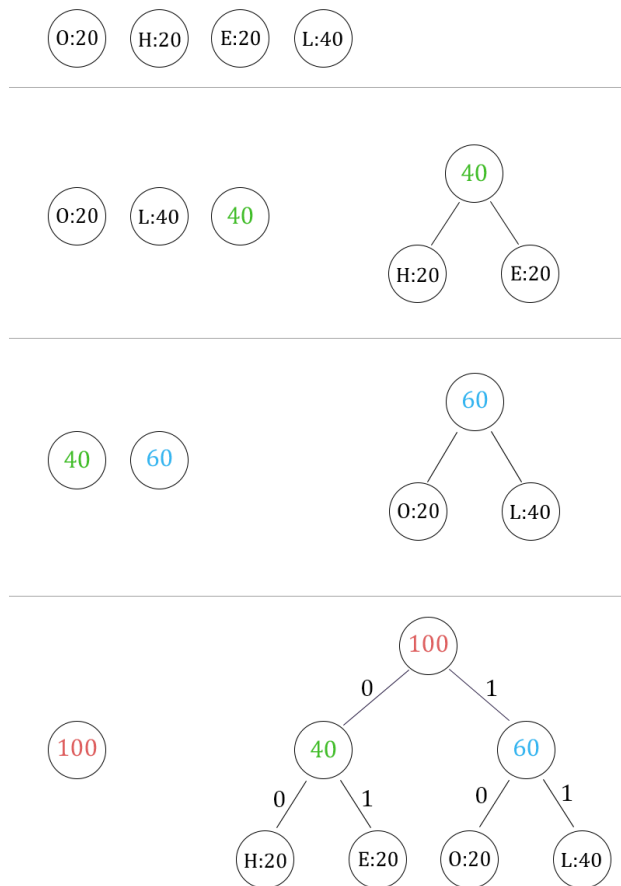
Se desea codificar la palabra "HELLO", donde cada carácter es asignado un código binario, de forma que el la cadena original sea representada en el menor número de bits posible.

Character	Appears	Frequency
H	1	25
e	1	25
l	2	50
o	1	25

Con lo anterior, se siguen los siguientes Pasos

1. Cada carácter representada un nodo con su frecuencia, formando un conjunto de nodos.
2. Los dos nodos con menos frecuencia son usados para crear un nuevo nodo cuya representación es la suma de la frecuencia de los nodos usados..
3. Los dos nodos usados son eliminados del conjunto de nodos y se agrega el nuevo nodo. Este proceso es repetido hasta que quede sólo un nodo, que representa la suma total de las frecuencias.
4. Se asigna a cada nodo un código binario dependiendo de su ubicación en el árbol: 0 para el sub-árbol izquierdo y un 1 para el sub-árbol derecho.

Con los anteriores Pasos, tenemos la construcción del árbol.



Con el árbol construido se puede observar el resultado de la codificación.

H: 00 E: 01 O: 10 L: 11

Por último, el mensaje "HELLO", es representado por:

0001111110

3 Implementación en Python

```
import heapq

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

    # for priority queue (frequency basically)
    def __lt__(self, other):
        return self.freq < other.freq

def huffman(frequencies):
    # Priority queue (min-heap)
    heap = [Node(char, freq) for char, freq in frequencies.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        # get 2 nodes with the smallest frequencies
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)

        # new node, the sum of the 2 nodes selected
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right

        # new node, so append it into the heap
        heapq.heappush(heap, merged)

    # final node = root of the huffman tree
    root = heap[0]

    # generating the codes
    codes = {}
    def generate_codes(node, current_code = ""):
        if node is None:
            return
        if node.char is not None: # leaf node
            codes[node.char] = current_code
            return
        generate_codes(node.left, current_code + "0")
        generate_codes(node.right, current_code + "1")

    generate_codes(root)
    return codes
```

Se hace uso de la librería *heapq* para crear la cola de prioridad *min-heap*, esto se logra con el método especial *lt* (*less than*) de la clase *Node*, que sirve para comparar instancias de esta clase. Y se necesita un *__lt__*.

Una *min-heap* es una estructura de datos con las siguientes propiedades:

- Es un árbol binario completo
- El valor de la raíz DEBE ser menor que todos sus nodos descendientes.

La *min-heap* es usada para priorizar los nodos con las frecuencias más bajas. La clase *Node* tiene atributos que representan el carácter y su frecuencia (dependiendo si es una hoja o no), y punteros a sus nodos descendientes (derecho e izquierdo); además de usar el método especial *less than* ya mencionado.

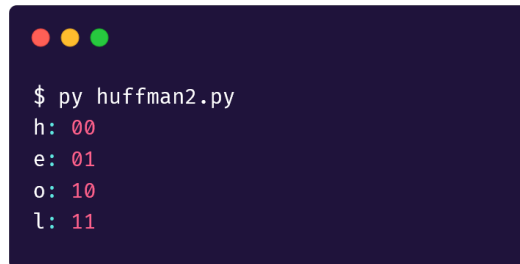
La función *huffman* crea la cola de prioridad, que se define como *heapq.heapify(heap)*, tomando como prioridad los nodos con menor frecuencia haciendo uso del método especial de la clase *Node*. Después crea el árbol empleando el método mencionado anteriormente. Por último, se define la función recursiva *generate codes* que asigna el respectivo código binario a los caracteres; Como caso base se tiene cuando el nodo es *None*, y cuando el nodo es una hoja (contiene un carácter) la función asigna el código actual a este carácter en el diccionario de códigos y se detiene en esa ruta.

4 Resultado

```
def main():
    frequencies = {
        'e': 20,
        'o': 20,
        'h': 20,
        'l': 40,
    }

    codes = huffman(frequencies)
    for i, j in codes.items():
        print(f"{i}: {j}")

main()
```

A terminal window with a dark blue background and three colored window control buttons (red, yellow, green) at the top left. The text inside the terminal is as follows:

```
$ py huffman2.py  
h: 00  
e: 01  
o: 10  
l: 11
```

Con el código ejecutado, podemos ver que una de las codificaciones válidas para la palabra "hello" es:

H: 00 E: 01 O: 10 L: 11

5 Time Complexity

La complejidad temporal de la construcción del árbol de Huffman depende del método utilizado para construirlo. El uso de una cola de prioridad para fusionar nodos tiene un Time Complexity de $O(n \log n)$, donde n es la cantidad de caracteres únicos en la entrada.

6 Comparación con ASCII

En ASCII se usa 8 bits para representar cada carácter, calculado en costo total de la palabra "hello" en ASCII se multiplica la frecuencia de cada carácter por 8 y después se suma.

$$Total = 1 \cdot 8 + 1 \cdot 8 + 1 \cdot 8 + 2 \cdot 8 = 40\text{bits}$$

Para esta representación de la palabra "hello" se mide de la siguiente manera:

1. Obtener la frecuencia de cada carácter
2. obtener la longitud del código asignado a cada carácter.
3. multiplicar el los dos anteriores y sumar.

Ejemplificando lo anterior con la palabra "hello":

- h: Freq: 1, Longitud: 2 (00)
- e: Freq: 1, Longitud: 2 (01)

- o: Freq: 1, Longitud: 2 (10)
- l: Freq: 2, Longitud: 2 (11)

Con esto, se obtiene:

$$\text{Total} = 2 \cdot 1 + 2 \cdot 1 + 2 \cdot 1 + 2 \cdot 2 = 10\text{Bits}$$

Lo que concluye a un ahorro significativo frente a la codificación ASCII.

7 Alogirtmo Voráz

La parte voraz del algoritmo de codificación de Huffman reside en el proceso de combinar repetidamente los dos nodos con las frecuencias más pequeñas, lo que en cada paso el algoritmo hace una elección óptima a nivel local (minimizando la frecuencia combinada de los nodos fusionados) con la esperanza de que esto conduzca a una solución óptima a nivel global (un árbol con la longitud de ruta ponderada mínima). Además, el algoritmo no mira hacia adelante ni intenta evaluar todas las combinaciones posibles para funcionar los nodos (se toma la decisión óptima a nivel local fusionando los dos nodos más pequeños en el paso actual).

8 Referencias

- GeeksforGeeks, “Introduction to MinHeap – Data Structure and Algorithm Tutorials,” GeeksforGeeks, Jun. 28, 2024. <https://www.geeksforgeeks.org/introduction-to-min-heap-data-structure/>
- G. Ershad, “HEAP (Priority Queue) — Identify Pattern - ITNEXT,” Medium, Aug. 23, 2024. [Online]. Available: <https://itnext.io/heap-priority-queue-identify-pattern-aaedda7b3f6b?gi=1e6a0da90ccb>
- Estudy, “Text Compression with Huffman Coding,” YouTube. Aug. 21, 2015. [Online]. Available: <https://www.youtube.com/watch?v=iiGZ947Tcck>
- GeeksforGeeks, “Time and space complexity of Huffman Coding algorithm,” GeeksforGeeks, Feb. 21, 2024. <https://www.geeksforgeeks.org/time-and-space-complexity-of-huffman-coding-algorithm/>