ETAPA III: Investigación sobre el algoritmo de Huffman

Ricardo Alexander López Hernández 00155321, Carlos César Portillo Mendoza 00155321

y Ricardo José Sibrián Rivera 00173821

Facultad de Ingeniería y Arquitectura, Universidad Centroamericana José Simeón Cañas

190161: Análisis de Algoritmos

Ing. Jorge Alfredo López Sorto

23 de noviembre de 2023

Algoritmo de Huffman

Es una técnica utilizada para la compresión de datos, es decir, busca reducir la cantidad de datos o el espacio de memoria para crear un mensaje.

En este algoritmo es necesario el uso de un alfabeto con un número de “n” caracteres finitos. Por lo que, debemos tener en cuenta que tipo de símbolos se podrán tener. Con esto, se sacan las frecuencias para calcular la probabilidad que contendrá cada letra en el mensaje y con ello se crea una secuencia de tipo binaria para cada una de ellas, que finalmente se unen para crear la nueva versión del mensaje.

Historia del algoritmo

En 1951, David Huffman se encontraba realizando sus estudios de doctorado en el Massachusetts Institute of Technology (MIT); mientras él cursaba la asignatura “Teoría de la información” el profesor Robert M. Fano les dio la opción a Huffman y sus demás compañeros de clase, de realizar un examen final o la presentación de un trabajo.

Huffman decidió realizar el trabajo, sin embargo , ante la posibilidad de demostrar que el código era más eficiente, este comenzó a rendirse por lo que empezó a estudiar para el examen final. No obstante mientras estaba en ese proceso le surgió la idea de usar árboles binarios de frecuencia ordenada y a partir de ello probó que este era el método más eficiente.

Con ello Huffman lograba superar a su mentor quien había trabajado con el inventor de la teoría de la información Claude Shannon; Con el fin de desarrollar un código similar; Huffman superó la mayor parte de errores en el algoritmo de codificación Shannon-Fano, su solución

estaba basada en el proceso de construir el árbol desde abajo hacia arriba en lugar de lo contrario desde arriba hacia abajo.

El algoritmo originalmente fue diseñado para codificar caracteres, sin embargo, más adelante se adoptó para comprimir datos a partir de la codificación de caracteres (Universidad de Sonsonate, 2017).

Explicación detallada del algoritmo

El algoritmo de Huffman provee un método que permite comprimir información mediante la recodificación de los bytes que la componen. En particular, si los bytes que se van a comprimir están almacenados en un archivo, al decodificarlos con secuencias de bits más cortas diremos que lo comprimimos. La técnica consiste en asignar a cada byte del archivo que vamos a comprimir un código binario compuesto por una cantidad de bits tan corta como sea posible. Esta cantidad será variable y dependerá de la probabilidad de ocurrencia del byte. Es decir: aquellos bytes que más veces aparecen serán recodificados con combinaciones de bits más cortas, de menos de 8 bits. En cambio, se utilizarán combinaciones de bits más extensas para recodificar los bytes que menos veces se repiten dentro del archivo. Estas combinaciones podrían, incluso, tener más de 8 bits.

Los códigos binarios que utilizaremos para reemplazar a cada byte del archivo original se llaman “códigos Huffman”

Veamos un ejemplo. En el siguiente texto:

COMO COME COCORITO COME COMO COSMONAUTA

El carácter ‘O’ aparece 11 veces y el carácter ‘C’ aparece 7 veces. Estos son los caracteres que más veces aparecen y por lo tanto tienen la mayor probabilidad de ocurrencia. En cambio, los caracteres ‘I’, ‘N’, ‘R’, ‘S’ y ‘U’ aparecen una única vez; esto significa que la probabilidad de hallar en el archivo alguno de estos caracteres es muy baja. Como ya sabemos, para codificar cualquier carácter se necesitan 8 bits (1 byte). Sin embargo, supongamos que logramos encontrar una combinación única de 2 bits con la cual codificar al carácter ‘O’, una

combinación única de 3 bits con la cual codificar al carácter ‘M’ y otra combinación única de 3 bits con la cual codifica al carácter ‘C’.

Byte o Carácter	Codificación
O	01
M	001
C	000

Si esto fuera así, entonces para codificar los primeros 3 caracteres del texto anterior solo necesitaríamos 1 byte, lo que nos daría una tasa de compresión del 66.6%.

Carácter	**C**	**O**	**M**	**…**
Byte	000	01	001	…

Ahora, el byte 00001001 representa la secuencia de caracteres ‘C’, ‘O’, ‘M’, pero esta información sólo podrá ser interpretada si conocemos los códigos binarios que utilizamos para recodificar los bytes originales. De lo contrario, la información no se podrá recuperar.

Para obtener estas combinaciones de bits únicas, el algoritmo de Huffman propone seguir una serie de pasos a través de los cuales obtendremos un árbol binario llamado “árbol Huffman”. Luego, las hojas del árbol representarán a los diferentes caracteres que aparecen en el archivo y los caminos que se deben recorrer para llegar a esas hojas representarán la nueva codificación del carácter.

A continuación, analizaremos los pasos necesarios para obtener el árbol y los códigos Huffman que corresponden a cada uno de los caracteres del texto expresado más arriba.

Paso 1 - Contar la cantidad de ocurrencias de cada carácter.

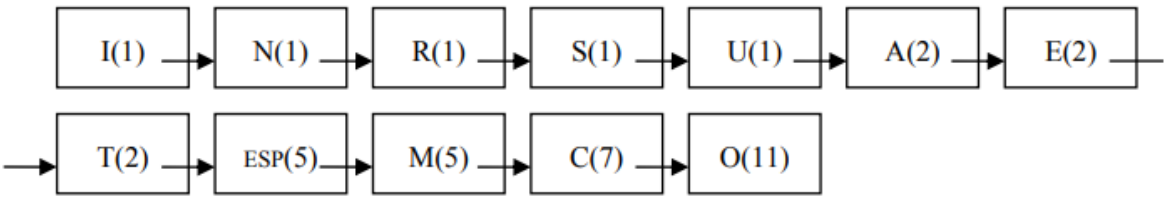
El primer paso consiste en contar cuántas veces aparece en el archivo cada carácter o byte. Como un byte es un conjunto de 8 bits, resulta que solo existen 28 = 256 bytes diferentes. Entonces utilizaremos una tabla con 256 registros para representar a cada uno de los 256 bytes y sus correspondientes contadores de ocurrencias.

	Carácter	*n*
0		
:		
32	*ESP*	5
:		
65	A	2
:		
67	C	7
:		
69	E	2
:		
73	I	1

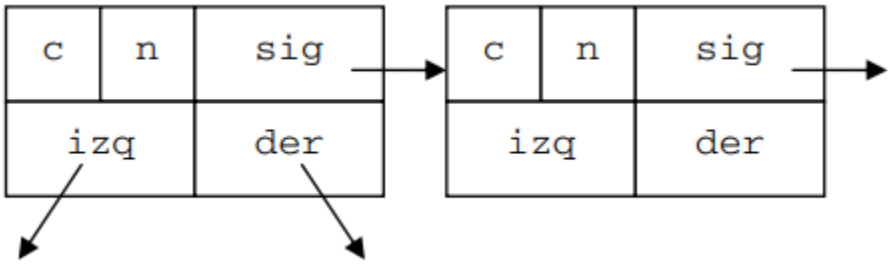
:	Carácter	*n*
:		
77	M	5
78	N	1
79	O	11
:		
82	R	1
83	S	1
84	T	2
85	U	1
:		
255		

Paso 2 - Crear una lista enlazada

Conociendo la cantidad de ocurrencias de cada carácter, tenemos que crear una lista enlazada y ordenada ascendentemente por dicha cantidad. Primero los caracteres menos frecuentes y luego los que tienen mayor probabilidad de aparecer y, si dos caracteres ocurren igual cantidad de veces, entonces colocaremos primero al que tenga menor valor numérico. Por ejemplo: los caracteres ‘I’, ‘N’, ‘R’, ‘S’ y ‘U’ aparecen una sola vez y tienen la misma probabilidad de ocurrencia entre sí; por lo tanto, en la lista ordenada que veremos a continuación los colocaremos ascendentemente según su valor numérico o código ASCII



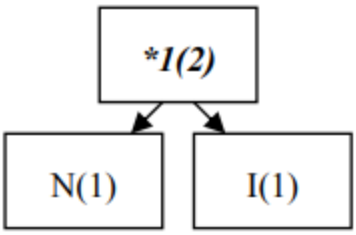
Los nodos de la lista tendrán la siguiente estructura:



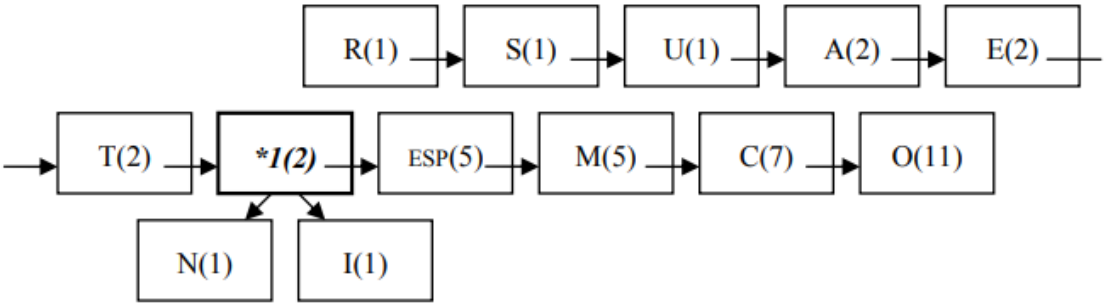
El campo c representa el carácter (o byte) y el campo n la cantidad de veces que aparece dentro del archivo. El campo sig es la referencia al siguiente elemento de la lista enlazada. Los campos izq y der, más adelante, nos permitirán implementar el árbol binario. Por el momento no les daremos importancia; simplemente pensamos que son punteros a null.

Paso 3 - Convertir la lista enlazada en el árbol Huffman.

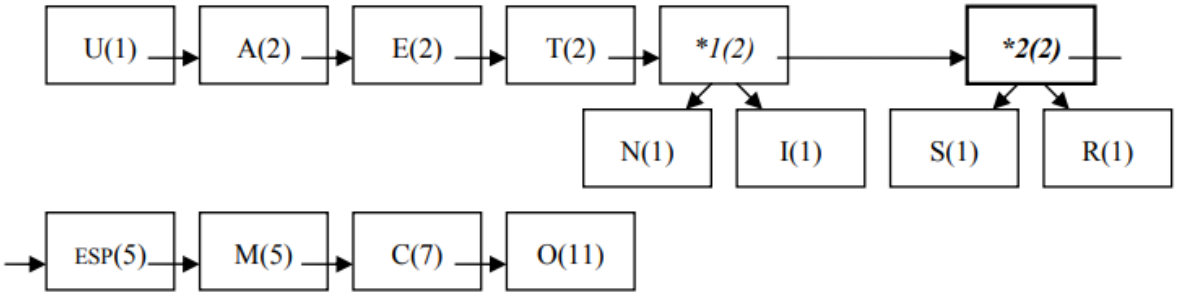
Vamos a generar el árbol Huffman tomando “de a pares” los nodos de la lista. Esto lo haremos de la siguiente manera: sacamos los dos primeros nodos y los utilizamos para crear un pequeño árbol binario cuya raíz será un nuevo nodo que identifi caremos con un carácter fi cticio *1 (léase “asterisco uno”) y una cantidad de ocurrencias igual a la suma de las cantidades de los dos nodos que estamos procesando. En la rama derecha colocamos al nodo menos ocurrente (el primero); el otro nodo lo colocaremos en la rama izquierda.



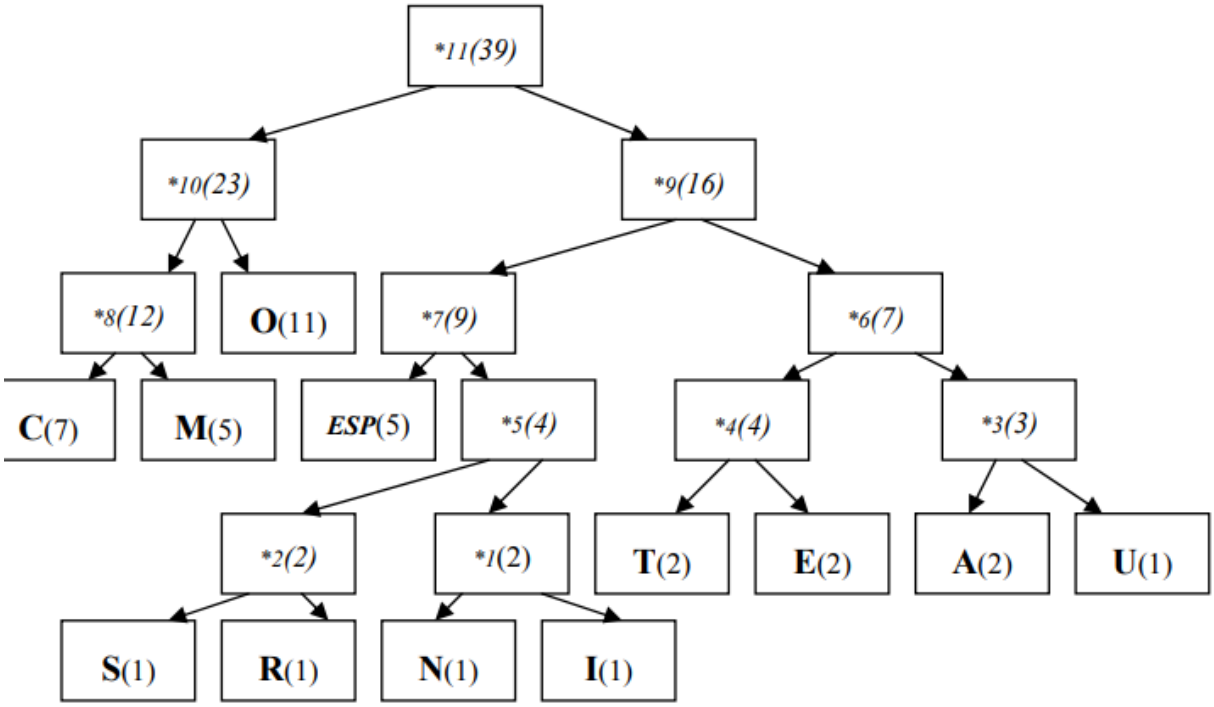
Luego insertamos en la lista al nuevo nodo (raíz) respetando el criterio de ordenamiento que mencionamos más arriba. Si en la lista existe un nodo con la misma cantidad de ocurrencias (que en este caso es 2), la inserción la haremos a continuación de este.



Ahora repetimos la operación procesando nuevamente los dos primeros nodos de la lista: R(1) y S(1):



Luego continuamos con este proceso hasta que la lista se haya convertido en un árbol binario cuyo nodo raíz tenga una cantidad de ocurrencias igual al tamaño del archivo que queremos comprimir.



Paso 4 - Asignación de códigos Huffman

El siguiente paso será asignar un código Huffman a cada uno de los caracteres reales que se encuentran ubicados en las hojas del árbol. Para esto, consideraremos el camino que se debe recorrer para llegar a cada hoja. El código se forma concatenando un 0 (cero) por cada tramo que avanzamos hacia la izquierda y un 1 (uno) cada vez que avanzamos hacia la derecha. Por lo tanto, el código Huffman que le corresponde al carácter ‘O’ es 01, el código que le corresponde al carácter ‘M’ es 001 y el código que le corresponde al carácter ‘S’ es 10100.

Como podemos ver, la longitud del código que el árbol le asigna a cada carácter es inversamente proporcional a su cantidad de ocurrencias. Los caracteres más frecuentes reciben códigos más cortos mientras que los menos frecuentes reciben códigos más largos. Agreguemos los códigos Huffman (cod) y sus longitudes (nCod) en la tabla de ocurrencias.

	Carácter	n	cod	nCod
0				
:				
32	ESP	5	100	3
:				
65	A	2	1110	4
:				
67	C	7	000	3
:				
69	E	2	1101	4
:				
73	I	1	10111	5

:	Carácter	n	cod	nCod
:				
77	M	5	001	3
78	N	1	10110	5
79	O	11	01	2
:				
82	R	1	10101	5
83	S	1	10100	5
84	T	2	1100	4
85	U	1	11111	5
:				
255				

Paso 5 - Codificación del contenido.

Para finalizar, generamos el archivo comprimido reemplazando cada carácter del archivo original por su correspondiente código Huffman, agrupando los diferentes códigos en paquetes de 8 bits ya que esta es la menor cantidad de información que podemos manipular.

C	O	M	O	*[esp]*	C	...
000	01	001	01	100	000	...
00001001			01100000			...

Paso 6 - Proceso de decodificación y descompresión.

Para descomprimir un archivo necesitamos disponer del árbol Huffman utilizado para su codificación. Sin el árbol la información no se podrá recuperar. Por este motivo el algoritmo de Huffman también puede utilizarse como algoritmo de encriptación. Supongamos que, de alguna manera, podemos rearmar el árbol Huffman, entonces el algoritmo para descomprimir y restaurar el archivo original es el siguiente: 1. Rearmar el árbol Huffman y posicionarnos en la raíz. 2. Recorrer “bit por bit” el archivo comprimido. Si leemos un 0 descendemos un nivel del árbol posicionándonos en su hijo izquierdo. En cambio, si leemos un 1 descendemos un nivel para posicionarnos en su hijo derecho. 3. Repetimos el paso 2 hasta llegar a una hoja. Esto nos dará la pauta de que hemos decodificado un carácter y lo podremos grabar en el archivo que estamos restaurando.

Ejemplos de aplicación

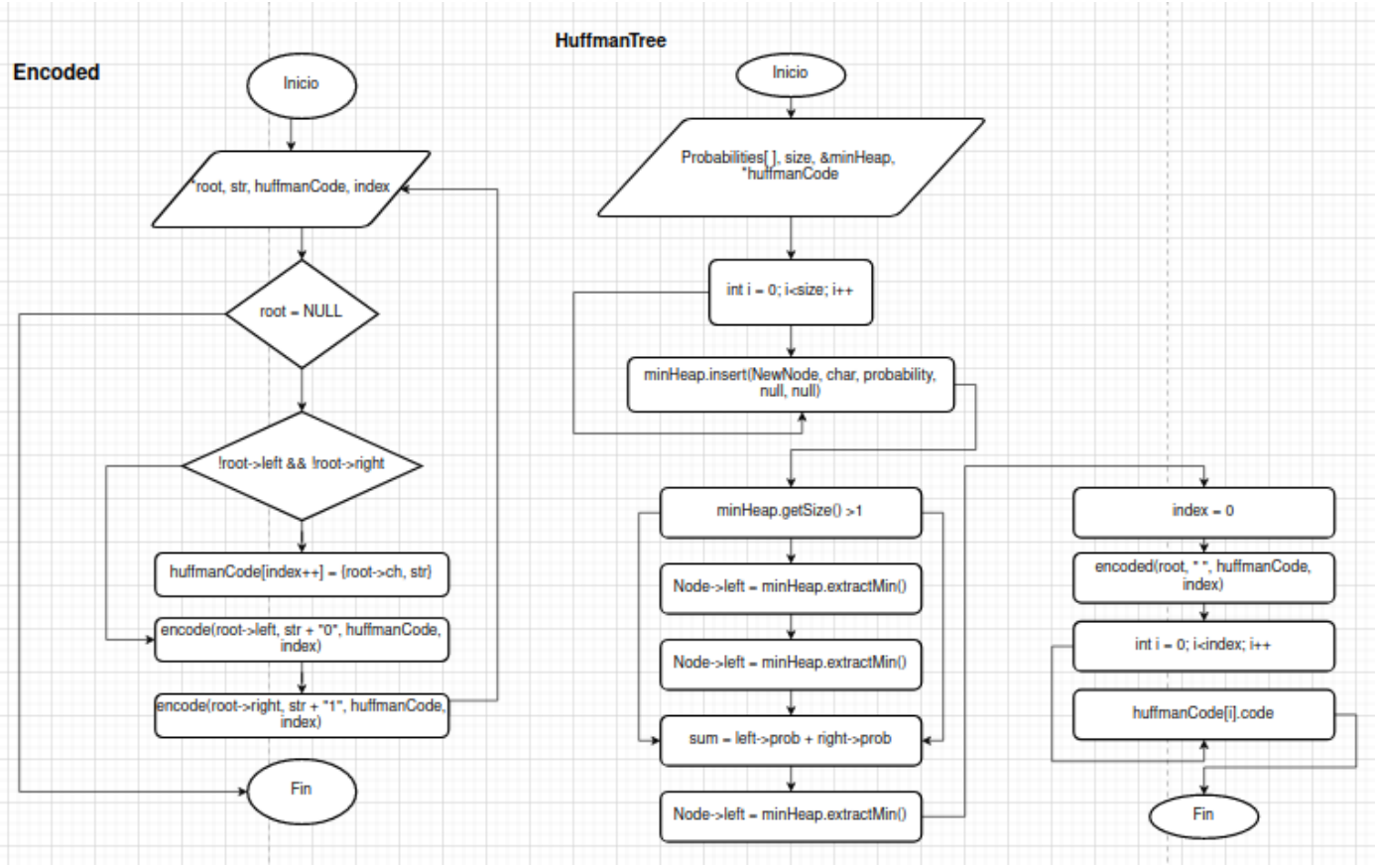
El algoritmo de Huffman es un algoritmo de compresión de datos que se utiliza para reducir el tamaño de los archivos. La idea básica detrás de la compresión de archivos es utilizar “codificación de longitud variable” para representar el mismo fragmento de texto utilizando un número menor de bits. En la codificación de longitud variable, asignamos un número variable de bits a los caracteres según su frecuencia en el texto dado. Entonces, algunos caracteres podrían terminar tomando un solo bit, y algunos podrían terminar tomando dos bits, algunos podrían codificarse usando tres bits, y así sucesivamente. El algoritmo de Huffman se utiliza para construir códigos de longitud variable que minimizan la longitud total de los códigos. Algunos ejemplos de aplicaciones del algoritmo de Huffman son: (Nonal, 2022)

- **Compresión de archivos de audio y video:** El algoritmo de Huffman se utiliza en la compresión de archivos de audio y video para reducir el tamaño de los archivos sin comprometer la calidad del sonido o la imagen.
- **Compresión de archivos de texto:** El algoritmo de Huffman se utiliza en la compresión de archivos de texto para reducir el tamaño de los archivos sin perder información.
- **Compresión de imágenes:** El algoritmo de Huffman se utiliza en la compresión de imágenes para reducir el tamaño de los archivos sin comprometer la calidad de la imagen.
- **Compresión de archivos ejecutables:** El algoritmo de Huffman se utiliza en la compresión de archivos ejecutables para reducir el tamaño de los archivos sin comprometer la funcionalidad del programa.
- **Compresión de archivos de bases de datos:** El algoritmo de Huffman se utiliza en la compresión de archivos de bases de datos para reducir el tamaño de los archivos sin comprometer la integridad de los datos.

Pseudocódigo del algoritmo

```
void encode(Node *root, string str, CharCode *huffmanCode, int &index)
{
        Inicio
        Si root = NULL terminamos la codificación
        Si root->left y root->right existen entonces:
            huffmanCode[index++] = {root->ch, str};

        Llamamos recursivamente la función encode(root->left, str + "0",
        huffmanCode, index)
        Llamamos recursivamente la función encode(root->right, str + "1",
        huffmanCode, index)
        Final
}


void buildHuffmanTree(const double probabilities[], int size, MinHeap &minHeap, CharCode
*huffmanCode)
{
    Inicio
    Para i = 0 hasta, mientras i < size hacer:
    insertmos en la cola(new Node{staticcast<char>('A' + i), staticcast<int>(probabilities[i]),
    nullptr, nullptr});
    Mientras minHeap.getSize() > 1 hacer:
        // Obtiene los mínimos de la cola de prioridad
        Node *left = minHeap.extractMin();
        Node *right = minHeap.extractMin();

        Sumamos las probabilidades de los nodos extraídos
        sum = left->prob + right->prob

    Node *root = minHeap.extractMin()
    Seteamos index como 0
    Llamamos la funcion encode(root, "", huffmanCode, index)
    Final
}
```

Diagrama de flujo del algoritmo



Análisis de eficiencia del algoritmo

Base Case

```
void encode(Node *root, string str, CharCode *huffmanCode, int &index)
{
        Inicio
        Si root = NULL terminamos la codificación  >>>>> En el caso base root = NULL O(1)
        Si root->left y root->right existen entonces: >>>> No se ejecuta ya que termina el if
            huffmanCode[index++] = {root->ch, str}; >>>> No se ejecuta ya que termina el if

        Llamamos recursivamente la función encode(root->left, str + "0",
        huffmanCode, index) >>>>>>>> En este caso no llega hasta aquí porque la raíz es nula
        Llamamos recursivamente la función encode(root->right, str + "1",
        huffmanCode, index) >>>>>>>> En este caso no llega hasta aquí porque la raíz es nula
        Final
}
```

En el caso base la complejidad temporal de la función queda en el orden de O(1)

```
void buildHuffmanTree(const double probabilities[], int size, MinHeap &minHeap, CharCode
*huffmanCode)
{
    Inicio
    Para i = 0 hasta, mientras i < size hacer: >> En el caso base la size = 0, comparación O(1) T0
        insertmos en la cola(new Node{staticcast<char>('A' + i), staticcast<int>(probabilities[i]),
        nullptr, nullptr}); >>> No entra el for
        Mientras minHeap.getSize() > 1 hacer: >>> Al no tener ningún nodo en la cola de
prioridad sale O(1)
                // Obtiene los mínimos de la cola de prioridad
                Node *left = minHeap.extractMin(); >>> No entra el while
                Node *right = minHeap.extractMin(); >>> No entra el while

                Sumamos las probabilidades de los nodos extraídos
                sum = left->prob + right->prob>>> No entra el while

        Node *root = minHeap.extractMin() >>> Al no tener ningún nodo en la cola de prioridad
sale O(1)
        Seteamos index como 0 >>> O(1) asignación T1
        Llamamos la funcion encode(root, "", huffmanCode, index) >> O(1) ver anterior análisis
T2
        Final
}

T0 + T1 + T2 = O(1)
En el caso base la complejidad temporal de la función queda en el orden de O(1)
```

Best Case

El mejor caso denota que el árbol de huffman está completamente balanceado!

```
void encode(Node *root, string str, CharCode *huffmanCode, int &index)
{
        Inicio
        Si root = NULL terminamos la codificación  >>>> O(1) ya que es una comparación y
        solo se ejecuta una vez T0
        Si !root->left y !root->right entonces: >>>> O(1) ya que es una comparación T1
            huffmanCode[index++] = {root->ch, str}; >>>> O(1) ya que es una asignación T2

        Llamamos recursivamente la función encode(root->left, str + "0",
        huffmanCode, index) >>>>>>>> Cómo está balanceado debe de ser O(n) T3
        Llamamos recursivamente la función encode(root->right, str + "1",
        huffmanCode, index) >>>>>>>> Cómo está balanceado debe de ser O(n) T4
        Final
```

```
}
```
T0 + T1 + T2 + T3 + T4 = O(log2(n))
En el mejor caso la complejidad temporal de la función queda en el orden de O(n)

```
void buildHuffmanTree(const double probabilities[], int size, MinHeap &minHeap, CharCode
*huffmanCode)
{
    Inicio
    Para i = 0 hasta, mientras i < size hacer: >>> En el mejor caso la size = 1, O(1) T0
        insertmos en la cola(new Node{staticcast<char>('A' + i), staticcast<int>(probabilities[i]),
        nullptr, nullptr}); >>> La insercion a un montículo tiene un costo de O(nlog2n) T1
        Mientras minHeap.getSize() > 1 hacer: >>> Al tener 1 solo nodo en la cola de prioridad
sale O(1) T2
                // Obtiene los mínimos de la cola de prioridad
                Node *left = minHeap.extractMin(); >>> No entra el while
                Node *right = minHeap.extractMin(); >>> No entra el while

                Sumamos las probabilidades de los nodos extraídos
                sum = left->prob + right->prob>>> No entra el while

        Node *root = minHeap.extractMin() >>> La función de extraer mínimo tiene un orden de
magnitud de O(nlog2n)
        Seteamos index como 0 >>> O(1) T4
        Llamamos la funcion encode(root, "", huffmanCode, index) >>> O(n) ver anterior T5
        Final
}
```

T0 + T1 + T2 + T3 + T4 +T5 = O(nlog2(n))
En el mejor caso la complejidad temporal de la función queda en el orden de O(nlog2(n))

Worst Case

El peor caso denota que el árbol se encuentre completamente desbalanceado!, curiosamente tiene el mismo orden de complejidad

```
void encode(Node *root, string str, CharCode *huffmanCode, int &index)
{
        Inicio
        Si root = NULL terminamos la codificación  >>>> O(1) ya que es una comparación y
      solo se ejecuta una vez T0
        Si !root->left y !root->right entonces: >>>> O(1) ya que es una comparación T1
            huffmanCode[index++] = {root->ch, str}; >>>> O(1) ya que es una asignación T2

        Llamamos recursivamente la función encode(root->left, str + "0",
        huffmanCode, index) >>>>>>>> Cómo está desbalanceado debe de ser O(n) T3
        Llamamos recursivamente la función encode(root->right, str + "1",
        huffmanCode, index) >>>>>>>> Cómo está desbalanceado debe de ser O(n) T4
        Final
}
T0 + T1 + T2 + T3 + T4 = O(n)
En el mejor caso la complejidad temporal de la función queda en el orden de O(n)

void buildHuffmanTree(const double probabilities[], int size, MinHeap &minHeap, CharCode
*huffmanCode)
{
   Inicio
   Para i = 0 hasta, mientras i < size hacer: >>> En el peor caso la size = n O(n) T0
      insertmos en la cola(new Node{staticcast<char>('A' + i), staticcast<int>(probabilities[i]),
      nullptr, nullptr}); >>> La inserción a un montículo tiene un costo de O(nlog2n) T1
      Mientras minHeap.getSize() > 1 hacer: >>> Como es una comparación O(n) T2
            // Obtiene los mínimos de la cola de prioridad
            Node *left = minHeap.extractMin(); >>> La función de extraer mínimo tiene un
orden de magnitud de O(nlog2n) T3
            Node *right = minHeap.extractMin(); >>> La función de extraer mínimo tiene un
orden de magnitud de O(nlog2n) T4

            Sumamos las probabilidades de los nodos extraídos
            sum = left->prob + right->prob>>>Es una suma O(1) T5

      Node *root = minHeap.extractMin() >>> La función de extraer mínimo tiene un orden de
magnitud de O(nlog2n) T6
      Seteamos index como 0 >>> O(1) T7
      Llamamos la función encode(root, "", huffmanCode, index) >>> O(n) ver anterior T8
      Final
}
```

T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 = O(nlog2n)
En el mejor caso la complejidad temporal de la función queda en el orden de O(nlog2n)

Conclusiones del análisis de eficiencia

1. La eficiencia del algoritmo depende enteramente de la cantidad de nodos que se insertan en el árbol de huffman, ya que utiliza un ciclo for para realizar el cuerpo completo de la función donde se utilizan las funciones de extraer el mínimo hasta que quede vacía la cola de prioridad, por lo que depende enteramente de la cantidad de elementos.
2. Es un algoritmo eficiente ya que en su peor caso y su mejor caso el orden de magnitud se encuentra estable.
3. Es un algoritmo fácil de entender y es muy eficiente para la codificación de textos lo cual lo hace un buen algoritmo.

Referencias

Martínez, L. (17 de febrero de 2019). *Alfabeto Arabe*. INFOALFABETOS. https://www.itespresso.es/oracle-adquiere-dyndns-158270.html

Moreno, C. (28 de abril de 2021). *Idioma árabe: sus características y su origen*. Tatutrad. https://tatutrad.net/idioma-arabe-sus-caracteristicas-y-su-origen

Encyclopædia Britannica. (13 de octubre de 2023). *Arabic alphabet*. https://www.britannica.com/topic/Arabic-alphabet

Tradulingua. (25 de octubre de 2022). *Idioma árabe: origen, características, curiosidades y por qué es importante interpretar a este idioma*. https://tradulingua.com/idioma-arabe/

AcademiaLab. (2023). *Historia del alfabeto árabe*. https://academia-lab.com/enciclopedia/historia-del-alfabeto-arabe/#google_vignette

Wikipedia. (2022, 26 febrero). *Codificación Huffman*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Codificaci%C3%B3n_Huffman

Universidad de Sonsonate. (2017). Algoritmo de Huffman. *Revista Integración*, 45. https://ui.usonsonate.edu.sv/papers/1972-1117_Revista_Integracion_2017.pdf#page=43

Nonal, C. (2022, septiembre 24). *Algoritmo de compresión de codificación de Huffman*. https://www.techiedelight.com/es/huffman-coding/