Pseudocódigo: Códigos de Huffman con Fuerza Bruta

Algoritmo de Fuerza Bruta

25 de septiembre de 2025

1. Descripción del Problema

El problema de codificación de Huffman consiste en encontrar el código de longitud variable óptimo para un conjunto de símbolos dados sus frecuencias de aparición. El objetivo es minimizar la longitud promedio de codificación, que se calcula como:

$$\text{Costo} = \sum_{i=1}^{n} f_i \cdot l_i$$

donde f_i es la frecuencia del símbolo i y l_i es la longitud del código asignado al símbolo i.

El algoritmo de fuerza bruta resuelve este problema generando todos los posibles árboles binarios completos con n hojas y evaluando todas las asignaciones posibles de símbolos a longitudes de código.

2. Pseudocódigo

Algorithm 1 Generación de Árboles Binarios Completos

Require: Número de hojas n
Ensure: Lista de todos los árboles binarios completos con n hojas

if $n = 1$ **then return** $[x]$ ▷ Árbol con una sola hoja
end if
$trees \leftarrow$ lista vacía
for $left\_size = 1$ to $n - 1$ **do**
    $right\_size \leftarrow n - left\_size$
    for cada $left$ en $generate\_trees(left\_size)$ **do**
        for cada $right$ en $generate\_trees(right\_size)$ **do**
            $trees.append((left, right))$
        end for
    end for
end forreturn $trees$

Algorithm 2 Cálculo de Longitudes de Código

Require: Árbol binario $tree$ y profundidad inicial $depth$
Ensure: Lista de longitudes de código para cada hoja

if $tree = x$ **then return** $[depth]$ ▷ Hoja del árbol
end if
$left, right \leftarrow tree$
$left\_lengths \leftarrow code\_lengths(left, depth + 1)$
$right\_lengths \leftarrow code\_lengths(right, depth + 1)$ **return** $left\_lengths + right\_lengths$

Algorithm 3 Códigos de Huffman con Fuerza Bruta

Require: Lista de símbolos $symbols$ y frecuencias $freqs$
Ensure: Costo mínimo y asignación óptima de longitudes de código

$n \leftarrow$ longitud de $symbols$
$best\_cost \leftarrow \infty$
$best\_assignment \leftarrow$ **null**
for cada $tree$ en $generate\_trees(n)$ **do**
    $lengths \leftarrow code\_lengths(tree)$
    for cada $perm$ en $permutations(range(n))$ **do**
        $cost \leftarrow \sum_{i=0}^{n-1} freqs[perm[i]] \cdot lengths[i]$
        if $cost < best\_cost$ **then**
            $best\_cost \leftarrow cost$
            $best\_assignment \leftarrow \{symbols[perm[i]] : lengths[i]$ para $i = 0$ a $n-1\}$
        end if
    end for
end forreturn $best\_cost, best\_assignment$

3. Explicación del Algoritmo

3.1. Enfoque de Fuerza Bruta

El algoritmo de fuerza bruta para códigos de Huffman funciona de la siguiente manera:

- **Generación de árboles:** Se generan todos los posibles árboles binarios completos con n hojas
- **Cálculo de longitudes:** Para cada árbol, se calculan las longitudes de código de todas las hojas
- **Permutaciones de asignación:** Se prueban todas las posibles asignaciones de símbolos a longitudes
- **Evaluación de costo:** Se calcula el costo total para cada asignación
- **Optimización:** Se mantiene la mejor asignación encontrada

La complejidad viene del hecho de que se generan todos los árboles binarios posibles y se prueban todas las permutaciones de asignación.

3.2. Estructura de Árboles Binarios Completos

Un árbol binario completo con n hojas tiene exactamente $n-1$ nodos internos. Para n símbolos, el número de árboles binarios completos diferentes es el $(n-1)$-ésimo número de Catalan:

$$C_{n-1} = \frac{1}{n}\binom{2(n-1)}{n-1}$$

Para cada árbol, las longitudes de código están determinadas por las profundidades de las hojas, y cada símbolo puede asignarse a cualquier hoja del árbol.

3.3. Complejidad

- **Tiempo:** $O(C_{n-1} \cdot n! \cdot n)$
    - $O(C_{n-1})$ para generar todos los árboles binarios completos
    - $O(n!)$ para generar todas las permutaciones de asignación
    - $O(n)$ para calcular el costo de cada asignación
- **Espacio:** $O(n)$ para almacenar la mejor asignación

Esta complejidad exponencial hace que el algoritmo sea impráctico para problemas grandes.

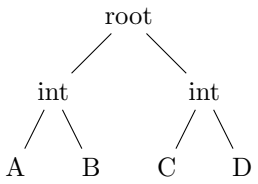
4. Resolución Paso a Paso

4.1. Ejemplo: Símbolos = [."A","B",Ç","D"], Frecuencias = [5,2,1,1]

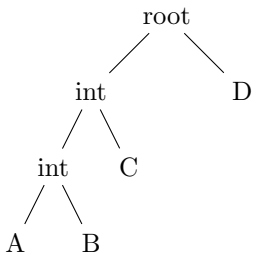
Paso 1: Identificar parámetros del problema

- Número de símbolos: $n = 4$
- Número de árboles binarios completos: $C_3 = \frac{1}{4}\binom{6}{3} = 5$
- Total de permutaciones por árbol: $4! = 24$
- Total de asignaciones a evaluar: $5 \times 24 = 120$

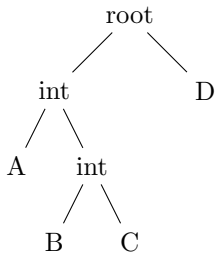
Paso 2: Generar todos los árboles binarios completos con 4 hojas
Los 5 árboles binarios completos son:
Árbol 1: ((x,x), (x,x))



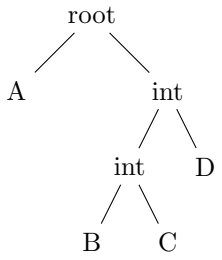
Longitudes: [2, 2, 2, 2]
Árbol 2: (((x,x), x), x)



Longitudes: [3, 3, 2, 1]
Árbol 3: ((x, (x,x)), x)

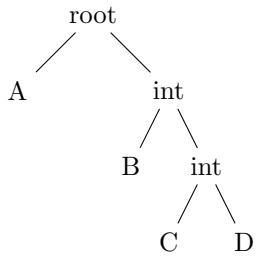


Longitudes: [2, 3, 3, 1]
Árbol 4: (x, ((x,x), x))



Longitudes: [1, 3, 3, 2]

Árbol 5: (x, (x, (x,x)))



Longitudes: [1, 2, 3, 3]

Paso 3: Evaluar asignaciones para cada árbol

Árbol 1: Longitudes [2, 2, 2, 2]

- Todas las asignaciones tienen el mismo costo: $5 \cdot 2 + 2 \cdot 2 + 1 \cdot 2 + 1 \cdot 2 = 18$
- **Costo:** 18

Árbol 2: Longitudes [3, 3, 2, 1]

- Asignación óptima: A→1, B→2, C→2, D→3
- Costo: $5 \cdot 1 + 2 \cdot 2 + 1 \cdot 2 + 1 \cdot 3 = 14$
- **Mejor hasta ahora:** 14

Árbol 3: Longitudes [2, 3, 3, 1]

- Asignación óptima: A→1, B→2, C→3, D→3
- Costo: $5 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 15$

Árbol 4: Longitudes [1, 3, 3, 2]

- Asignación óptima: A→1, B→3, C→3, D→2
- Costo: $5 \cdot 1 + 2 \cdot 3 + 1 \cdot 3 + 1 \cdot 2 = 16$

Árbol 5: Longitudes [1, 2, 3, 3]

- Asignación óptima: A→1, B→2, C→3, D→3
- Costo: $5 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 15$

Paso 4: Verificar si existe una mejor asignación en el Árbol 2
Evaluando todas las permutaciones del Árbol 2 con longitudes [3, 3, 2, 1]:

Permutación	A	B	C	D
[0, 1, 2, 3]	3	3	2	1
[0, 1, 3, 2]	3	3	1	2
[0, 2, 1, 3]	3	2	3	1
[0, 2, 3, 1]	3	2	1	3
[0, 3, 1, 2]	3	1	3	2
[0, 3, 2, 1]	3	1	2	3
[1, 0, 2, 3]	3	3	2	1
[1, 0, 3, 2]	3	3	1	2
[1, 2, 0, 3]	2	3	3	1
[1, 2, 3, 0]	2	3	1	3
[1, 3, 0, 2]	2	1	3	3
[1, 3, 2, 0]	2	1	3	3
[2, 0, 1, 3]	3	3	2	1
[2, 0, 3, 1]	3	3	1	2
[2, 1, 0, 3]	2	3	3	1
[2, 1, 3, 0]	2	3	1	3
[2, 3, 0, 1]	2	1	3	3
[2, 3, 1, 0]	2	1	3	3
[3, 0, 1, 2]	3	3	2	1
[3, 0, 2, 1]	3	3	1	2
[3, 1, 0, 2]	2	3	3	1
[3, 1, 2, 0]	2	3	1	3
[3, 2, 0, 1]	2	3	3	1
[3, 2, 1, 0]	2	3	1	3

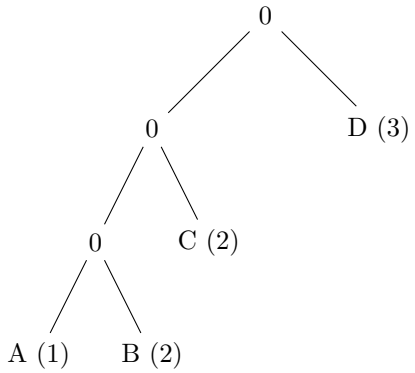
Calculando costos para las asignaciones más prometedoras:

- A→1, B→2, C→2, D→3: Costo $= 5 \cdot 1 + 2 \cdot 2 + 1 \cdot 2 + 1 \cdot 3 = 14$
- A→1, B→2, C→3, D→3: Costo $= 5 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 + 1 \cdot 3 = 15$

Resultado Final:

- **Mejor costo:** 14
- **Asignación óptima:** {A: 1, B: 2, C: 2, D: 3}
- **Árbol óptimo:** Árbol 2 con la asignación específica

4.2. Visualización del Árbol Óptimo



5. Análisis de Complejidad

5.1. Comparación con Algoritmo de Huffman Tradicional

- **Fuerza Bruta:** $O(C_{n-1} \cdot n! \cdot n)$ - Garantiza solución óptima
- **Huffman Tradicional:** $O(n \log n)$ - Solución óptima en tiempo polinomial

Ventajas del enfoque de fuerza bruta:

- Garantiza encontrar la solución óptima
- Útil para verificar la corrección del algoritmo de Huffman
- Apropiado para problemas pequeños

Desventajas:

- Complejidad exponencial
- Ineficiente para problemas grandes
- Redundante cuando existe el algoritmo óptimo de Huffman

6. Conclusiones

El algoritmo de fuerza bruta para códigos de Huffman garantiza encontrar la solución óptima pero tiene una complejidad exponencial que lo hace impráctico para problemas grandes. Es útil para:

- Verificar la corrección del algoritmo de Huffman tradicional
- Resolver instancias pequeñas del problema
- Estudios teóricos sobre la optimalidad de soluciones

Para aplicaciones prácticas, el algoritmo de Huffman tradicional (con complejidad $O(n \log n)$) es la opción recomendada, ya que produce la misma solución óptima de manera mucho más eficiente.