Algoritmos y Estructuras de Datos II
Clase 9

Carreras:

Licenciatura en Informática

Ingeniería en Informática

2024

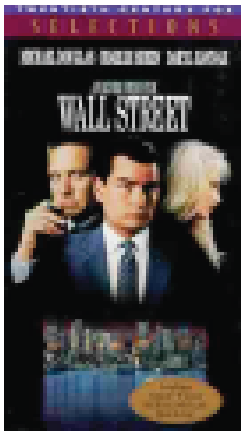
Unidad III

Técnicas de diseño de algoritmos

Algoritmos greedy (1)

Greed

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all forms, greed for life, money, love, knowledge has marked the upward surge in mandkind.



-Michael Douglas como Gordon Gekko ,

Wall Street (1987)

Significado de Greedy

Concise Oxford Spanish Dictionary © 2009 Oxford University Press:

greedy / ˈgriːdi/ *adjetivo* **-dier, -diest**

(for food, drink) glotón, angurriento (CS);

(for power, wealth) **to be ~ FOR sth** tener ansias or estar ávido de algo

Diccionario Espasa Concise © 2000 Espasa Calpe:

greedy ['grɪ:dɪ] *adj (**greedier, greediest**)*

1 *(de comida)* glotón,-ona

2 *(de dinero)* codicioso,-a [**for,** de]: **he is greedy for money,** ansía el dinero

3 *(de poder)* ávido,-a

Algoritmo Greedy

Algoritmo que es por naturaleza: voraz, ávido, codicioso, ambicioso, goloso, glotón, comilón, angurriento, acaparador, ansioso, etc etc.

El nombre de algoritmo *greedy* se debe a su comportamiento:

- En cada etapa "toman lo que más les conviene" sin analizar consecuencias.
- Avanzan siempre hacia "la solución más prometedora".
- La Técnica Greedy para el diseño de algoritmos es muy poderosa y funciona bien para un amplio espectro de aplicaciones.

Algoritmo Greedy

Los algoritmos greedy son:

- sencillos,
- fáciles de inventar,
- fáciles de implementar,
- cuando funcionan, son eficientes.
- generalmente utilizan la estrategia top-down

- Se usan para resolver *problemas de optimización*.

Algoritmo Greedy

La idea del enfoque greedy en los algoritmos, sugiere la construcción de la solución a lo largo de una secuencia de pasos, cada uno de los cuales expande la solución parcial obtenida, hasta que se alcanza la solución del problema.

En cada paso lo central de la técnica es que **la elección** debe ser:

- *Factible*, debe satisfacer las restricciones del problema.
- *Localmente optima*, debe ser la mejor elección local entre todas las alternativas disponibles en ese paso.
- *Irrevocable*, una vez que se hace una elección la decisión es irreversible y no puede ser cambiada en los pasos siguientes del algoritmo.

Algoritmo Greedy

Se aplica a problemas de optimización que se pueden resolver mediante una ***secuencia de decisiones*.**

Un algoritmo greedy:

- trabaja en una secuencia de pasos
- en cada etapa hace una elección local que se considera una *decisión óptima*,
- luego se resuelve el subproblema que resulta de esta elección,
- finalmente estas soluciones localmente óptimas se integran en una solución global óptima.

Para cada problema de optimización se debe demostrar que la elección óptima en cada paso, lleva a una solución óptima global.

Frecuentemente se preprocesa la Entrada o se usa una una Estructura de Datos adecuada para hacer rápida la elección greedy y así tener un algoritmo más eficiente.

Algoritmo Greedy

- Dado un problema con n entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema.

- Cada uno de los subconjuntos que cumplan las restricciones se dice que son *soluciones prometedoras.*

- Una solución prometedora que *maximice* o *minimice* una *función objetivo* se denomina *solución óptima*.

- Cuando el algoritmo greedy funciona correctamente, *la primera solución que encuentre es siempre óptima.*

Algoritmo Greedy

Elementos que deben estar presentes en el problema:

- Un conjunto de *candidatos*, que corresponden a las *n* entradas del problema.
- Una función que compruebe si un subconjunto de estas entradas es *solución* al problema, sea óptima o no.
- Una *función de selección* que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor* .
- Una *función objetivo* que determine el valor de la solución hallada. Es la función que se quiere maximizar o minimizar.

Algoritmo Greedy

En un **algoritmo Greedy** se tiene generalmente:

C: conjunto de candidatos que nos sirven para construir la solución del problema.

S: conjunto de los candidatos ya usados para armar la solución

solución: una función que chequea si un conjunto es solución del problema, ignorando en principio si esta solución es óptima o no.

factible: una función que chequea si un conjunto de candidatos es factible como solución del problema sin considerar si es óptima o no.

selección: una función que indique cual es el candidato mas prometedor que no se eligió todavía. Está relacionada con la función objetivo.

objetivo: una función que es lo que se trata de optimizar. (Esta función no aparece en el algoritmo).

Esquema Algoritmo Greedy

```
FUNCIÓN Greedy (C): conjunto → conjunto
    S ← ∅
    MIENTRAS not solución (S) AND C ≠ ∅ HACER
        x ← selección (C)
        C ← C - {x}
        SI factible ( S U {x} ) ENTONCES
            S ← S U {x}
    FIN MIENTRAS
    SI solución (S) ENTONCES
        RETORNA (S)
    SINO
        RETORNA (∅ )
FIN
```

¿ Funcionan siempre?

En los algoritmos Greedy el proceso no finaliza al diseñar e implementar el algoritmo que resuelve el problema en consideración.

Hay una tarea extra muy importante:

- **SI el algoritmo FUNCIONA BIEN**: la ***demostración formal*** de que el algoritmo encuentra la solución óptima en todos los casos.

- **SI el algoritmo NO FUNCIONA**: la presentación de un ***contraejemplo*** que muestra los casos en donde falla.

Algoritmo Greedy

Pasos para diseñar un algoritmo greedy para resolver un problema:

- Modelar el problema de optimización como un problema en el que se hace una elección y resulta un subproblema para resolver.

- Probar que siempre existe una solución optima al problema original que hace esta elección greedy, de modo que la elección greedy es segura.

- Demostrar que, después de hacer la elección greedy, lo que queda es un subproblema con la propiedad de que si se combina la solución optima al subproblema con la elección greedy que se ha realizado, se llega a una solución optima del problema original.

Algoritmo Greedy

Como saber si un algoritmo greedy resolverá un problema particular de optimización?

No hay una regla general, pero hay dos ingredientes claves:

- La propiedad de elección greedy
- La subestructura optima.

Algoritmo Greedy

La propiedad de elección greedy

- Se puede llegar a una solución global optima haciendo elecciones locales optimas.

- Cuando se considera cual es la elección que conviene hacer se decide la elección que parece “la mejor” para el problema en consideración, sin considerar los resultados de los subproblemas, ni las futuras elecciones. Luego se resuelve el subproblema resultante de haber hecho esa elección.

- Por supuesto se debe probar que la elección greedy en cada paso lleva a la solución globalmente optima.

Algoritmo Greedy

Subestructura optima

- Un problema tiene subestructura optima si una solución optima del problema contiene en su interior soluciones optimas a subproblemas.
- Generalmente se usa un enfoque directo en lo que respecta subestructura optima cuando se la aplica a algoritmos greedy.
- Se puede suponer que se llega a un subproblema habiendo hecho una elección greedy en el problema original.
- Lo que realmente hay que sostener es que un solución optima al subproblema, combinada con la elección greedy ya efectuada, lleva a una solución optima del problema original.
- Este esquema implica el uso de la inducción en los subproblemas para demostrar que con una elección greedy en cada paso se produce una solución optima.

Algoritmos Greedy

Ejemplos ya conocidos:

- Dar cambio a un cliente con el número mínimo de monedas.
- Procesamiento de tareas, de modo de tener tiempo de espera mínimo.
- Problema de carga.
- Algoritmo de Dijkstra para encontrar los caminos mínimos de un vértice a los restantes del grafo.
- Algoritmo de Prim para encontrar el árbol de recubrimiento mínimo en un grafo.

Problema del cambio

Se quiere **dar vuelto a un cliente**, usando el mínimo número de monedas.

- Los candidatos: un conjunto de monedas representando los distintos valores. El conjunto contiene monedas suficientes de cada clase.
- una solución: el valor total del conjunto elegido de monedas es exactamente el valor a pagar.
- un conjunto factible: el valor total del conjunto elegido no excede el monto a pagar.
- la función objetivo: el numero de monedas usadas en la solución sea mínimo.
- la función de selección: elige la moneda de mas alto valor que queda en el conjunto de candidatos.

Problema del cambio

Funcion darvuelto(vuelto): entero → conjunto de monedas

C ← {25, 10, 5, 1} con suficiente cantidad de c/u

S ← ∅

suma ← 0

MIENTRAS suma ≠ vuelto HACER

x ← la moneda de mayor valor en C

C ← C - {x}

SI suma+x ≤ vuelto ENTONCES

S ← S U {x}

suma ← suma+x

FIN MIENTRAS

RETORNA S

FIN

Problema del cambio

- Los valores del algoritmo corresponden a las monedas mas usadas en USA: 1(penny), 5(nickel), 10(dime), y 25(quarter). Se puede demostrar que con estos valores de monedas (si hay disponibles de todas las denominaciones de monedas en cantidad suficiente en el conjunto inicial) *este algoritmo Greedy* que elige en cada paso la moneda de mayor valor***, siempre encontrará la solución optima.***
- Se puede demostrar que con los valores de monedas de los circulantes en Argentina, el *algoritmo Greedy* ***siempre encontrará la solución optima*** si es que existe una solución.
- En Europa las monedas son de valor: {1, 2, 5, 10, 20, 50 } centavos de euro y de 1€ y 2€, el *algoritmo Greedy* ***siempre encontrará la solución optima*** si es que existe una solución.

Problema del cambio

En el Reino Unido las monedas son de valor:

{1, 2, 5, 10, 20, 25, 50 } peniques (centavos de libra esterlina)

y de 1, 2 y 5 £ (libra esterlina)

Ejemplo:

Para pagar 40 peniques:

Solución greedy usa: 25 + 10 + 5 (3 monedas)

Solución óptima usa: 20 + 20 (2 monedas)

Con estos valores de monedas no funciona el algoritmo Greedy

Problema del cambio

En Mundo Binario las monedas tienen valor de 2^i, i=0,1,2,3,…k.

Se puede demostrar que el algoritmo greedy para dar vuelto que elige en cada paso la moneda de mayor valor da la solución optima, si es que existe cantidad suficiente de cada una.

Ideas:

El *vuelto* encontrado por la solución greedy se puede expresar como:

$$vuelto = x_0 + 2x_1 + 2^2 x_2 + \ldots + 2^n x_n$$

Sea $X = (x_0, x_1, \ldots, x_n)$ la solución encontrada por la técnica greedy.

Donde los x_i son *0 o 1.*

Se verifica además que n es el menor entero tal que: $vuelto < 2^{n+1}$

Problema del cambio

Se puede considerar otra solución Y obtenida por otra técnica.

$$Y = (y_0, y_1, \ldots, y_m)$$

De modo que el vuelto se puede escribir como:

$$vuelto = y_0 + 2y_1 + 2^2 y_2 + \ldots + 2^m y_m$$

Hay que demostrar que: $\sum_{i=0}^{n} x_i < \sum_{i=0}^{m} y_i$

Como se verifica que $vuelto < 2^{n+1}$ implica que $m \leq n$

Entonces: $y_{m+1} = y_{m+2} = \ldots = y_n = 0$

Con algunos pasos más se puede demostrar que:

$$x_0 + x_1 + \ldots + x_n < y_0 + y_1 + \ldots + y_n$$

Entonces la cantidad de monedas obtenidas en la técnica greedy es mínima y esa solución resulta ser la óptima.

Problema del cambio

Las ideas del Mundo Binario se pueden generalizar cuando se tiene que dar vuelto y las monedas tienen valor de: $p^i,\ i=0,1,2,3,...k.$, donde p es un número natural mayor que 1.

Se puede demostrar que el algoritmo greedy que elige en cada paso la moneda de mayor valor da la solución optima, si es que existe cantidad suficiente de cada una.

En este caso el *vuelto* encontrado por la solución greedy se puede expresar como:

$$vuelto = x_0 + p.x_1 + p^2.x_2 + \ldots + p^n.x_n$$

La solución encontrada por la técnica greedy será: $X = (x_0, x_1, \ldots, x_n)$

Las componentes x_i , $0 \leq i \leq n$ son valores en el rango: $0 \leq x_i < p$.

Se verifica además que n es el menor entero tal que : $vuelto < p^{n+1}$

Problema del cambio

Suponer ahora que, en el problema dar cambio con monedas, se dispone de un sistema monetario en el que cada tipo de moneda vale más del doble que el tipo anterior.

La moneda de menor valor tiene valor 1.

Se puede demostrar que estas condiciones no garantizan que el algoritmo greedy para dar el cambio con estas monedas encuentre siempre la solución óptima.

Problema de Carga

Un barco se tiene que cargar con contenedores. Los contenedores son todos del mismo tamaño, pero de distintos pesos. La capacidad de carga del barco esta prefijada.
Se quiere cargar el barco con el ***máximo número de contenedores.***

Datos: n contenedores numerados: $i=1,2,3,\ldots n$

p_i : peso de cada contenedor i,

M : capacidad máxima de carga del barco

Solución: vector X

$x_i = 0$ si el contenedor i no se carga en el barco

$x_i = 1$ si el contenedor i si se carga en el barco

Maximizar la cantidad: $\sum_{i=1}^{n} x_i$ **Restricción**: $\sum_{i=1}^{n} p_i x_i \leq M$

SOLUCIÓN: Estrategia greedy en peso:
en cada paso elegir el contenedor ***de menor peso posible*.**

Algoritmos Greedy

Ejemplos nuevos de aplicación de la técnica:

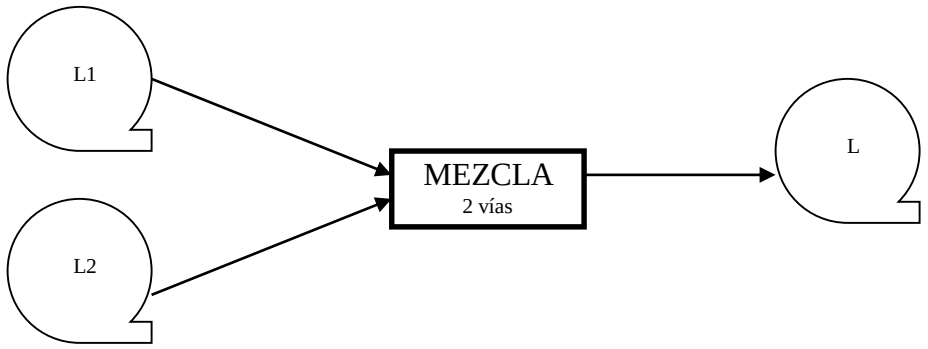
- Mezcla de 2 vías.
- Código de Huffman.
- Problema de la mochila (knapsack problem).
- Planificación (un servidor, varios servidores, con costo y beneficio).
- Asignación de tareas.
- Problema de grabación de archivos.
- Multiplicación óptima de matrices.

Problema de Mezcla de 2 vías

Suponga dos ***listas ordenadas*** L1 y L2 que se tienen que mezclar con un algoritmo de ***mezcla lineal***.

Ideas para Armar una nueva lista L que mezcle L1 y L2

- Hacer un recorrido secuencial de L1 y de L2
- En cada paso elegir el menor de los elementos
- Agregarlo a la lista L



Algoritmo Mezcla (L1, L2) :lista ordenada x lista ordenada→ lista ordenada

```
i←1 ;  j←1 ;  L←Listavacia
Hacer
        Si L2(j)<L1(i) entonces
                Agregar (L,L2(j) ) ;        j←j+1
        sino
                Agregar (L,L1(i) ) ;        i←i+1
Mientras (i≤Long(L1)) and (j≤Long(L2))
Si i>Long(L1) entonces
        Hacer
                Agregar (L,L2(j) ) ;        j←j+1
        Hasta que j=Long(L2)
Sino
        Hacer
                Agregar (L,L1(i) ) ;        i←i+1
        Hasta que i=Long(L1)
Retorna L
Fin
```

Mezcla de 2 vías

Algoritmo de Mezcla de 2 vías

Dos listas ordenadas L1 y L2 que mezclan con un algoritmo de mezcla lineal. Las listas L1 y L2 tienen n1 y n2 elementos respectivamente.

Costo del algoritmo de mezcla de 2 vías:

- La lista L1 tiene n1 elementos
- La lista L2 tiene n2 elementos
- Número de comparaciones: **n1+n2-1** en el peor caso.

Este algoritmo es óptimo.

Mezcla de 2 vías

Si se requiere mezclar *más de dos listas ordenadas* se puede usar este mismo algoritmo mezclando repetidamente las listas de a 2.

Así en varias aplicaciones del algoritmo se tendrá la lista ordenada.

Ahora se plantea el siguiente problema: suponga que se tiene m listas ordenadas, cada una de distinta cantidad de elementos.

¿ Cuál será la sucesión óptima del proceso de mezcla para mezclar todas estas listas ordenadas usando el *mínimo número de comparaciones*?

Mezcla de 2 vías

Ejemplo: Sean las 5 listas: (L1,L2,L3,L4,L5) con tamaños: (20,5,8,7,4). Estas 5 listas se pueden mezclar de distintas maneras.

A)

L1 se mezcla con **L2** para obtener **Z1**(tamaño 25)

→ **20+5-1=24 comparaciones**

Z1 se mezcla con **L3** para obtener **Z2**(tamaño 33)

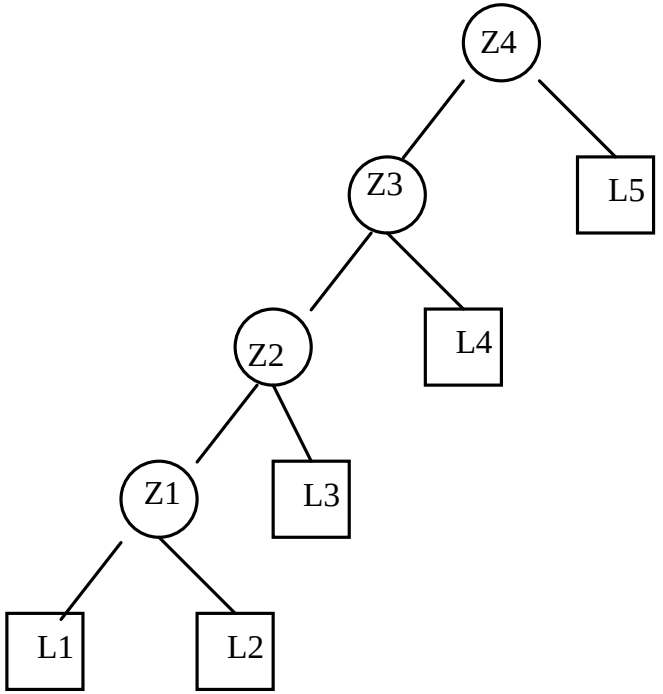
→ **25+8-1=32 comparaciones**

Z2 se mezcla con **L4** para obtener **Z3**(tamaño 40)

→ **33+7-1=39 comparaciones**

Z3 se mezcla con **L5** para obtener **Z4**(tamaño 44)

→ **40+4-1=43 comparaciones**



Total=138 comparaciones

Mezcla de 2 vías

B)

L2 se mezcla con **L5** para obtener **Z1** (tamaño 9)

→ **5+4-1=8 comparaciones**

L3 se mezcla con **L4** para obtener **Z2** (tamaño 15)

→ **8+7-1=14 comparaciones**

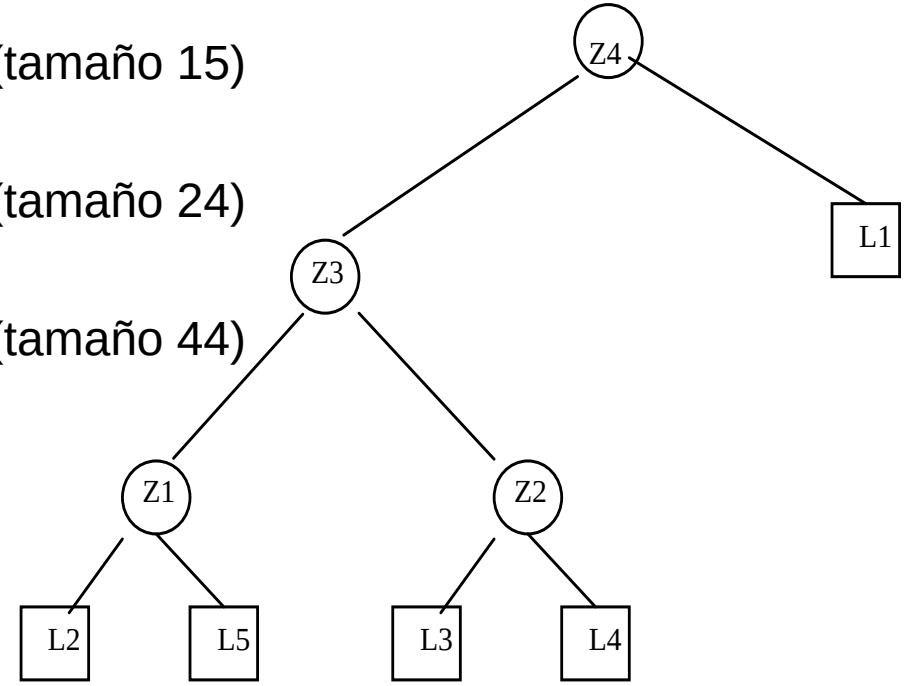
Z1 se mezcla con **Z2** para obtener **Z3** (tamaño 24)

→ **9+15-1=23 comparaciones**

Z3 se mezcla con **L1** para obtener **Z4** (tamaño 44)

→ **24+20-1=43 comparaciones**

Total=88 comparaciones



Mezcla de 2 vías

La idea es resolver este problema de mezclar m listas con el minimo de comparaciones con una estrategia greedy.

Se comienza mezclando las listas más cortas y *en cada paso se elige las dos de **menor longitud** para mezclar*.

Ejemplo: aplicar esta estrategia para generar el árbol de mezcla.

Los candidatos son las listas con sus respectivas longitudes:

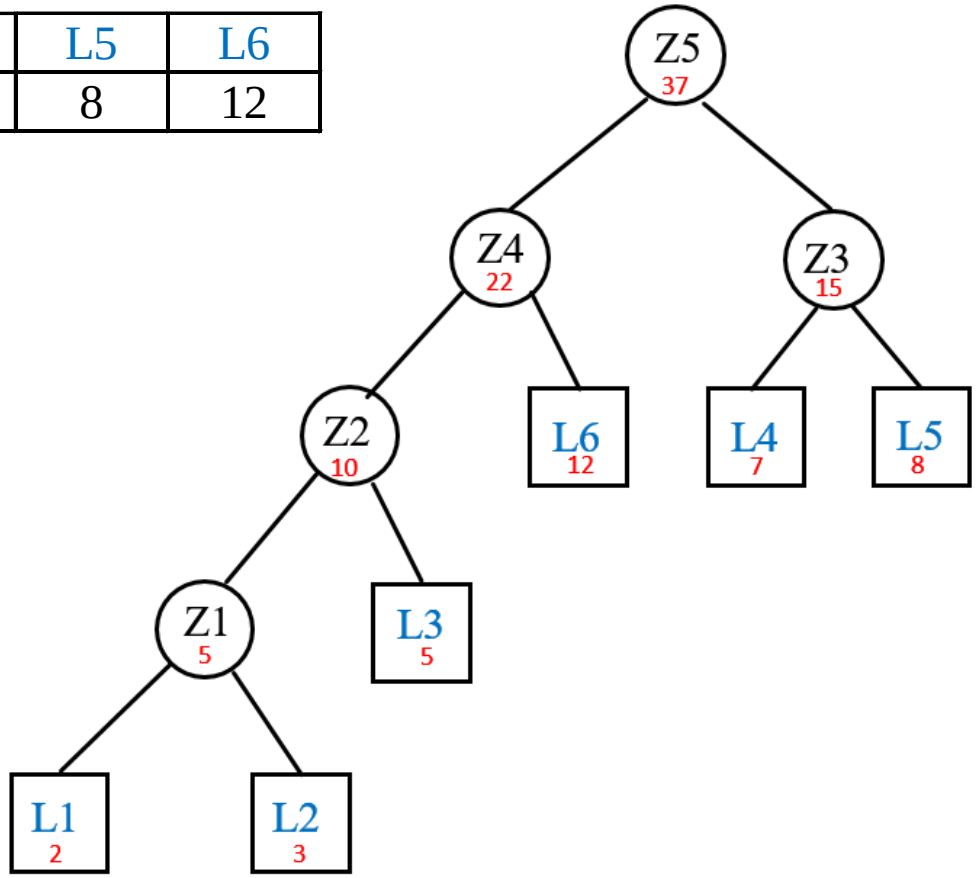
L1	L2	L3	L4	L5	L6
2	3	5	7	8	12

Mezcla de 2 vías

Ejemplo: aplicar esta estrategia para generar el árbol de mezcla.

Los candidatos son las listas con sus respectivas longitudes:

L1	L2	L3	L4	L5	L6
2	3	5	7	8	12



Mezcla de 2 vías

Se debe también demostrar que el algoritmo greedy para generar el árbol de mezcla de 2 vías da un resultado óptimo.

Complejidad temporal del algoritmo:

Sean las listas: $L_1, L_2, \ldots, L_m$

Sus respectivas longitudes: $n_1, n_2, \ldots, n_m$.

Para los m valores de las longitudes se construye un PQ de mínimo que se implementa con un heap.

Cada inserción en el heap se hace en tiempo $O(\log_2 m)$, lo mismo que cuesta cada borrado de la raíz.

La iteración se realiza m-1 veces.

El tiempo total de armar el árbol óptimo para mezclar m listas:

$$\mathbf{T(m) \in O(m.\log_2 m)}$$

Codificación

- La ***compresión de datos*** consiste en la transformación de una cadena de caracteres de cierto alfabeto en una nueva cadena que contiene la misma información, pero cuya longitud es menor que la de la cadena original.
- Esto requiere el diseño de un ***código*** que pueda ser utilizado para representar de manera única todo carácter de la cadena de entrada.
- El proceso de transformación del mensaje fuente en código se conoce como ***codificación***.
- El proceso inverso se denomina ***decodificación***.

Codificación
Códigos de longitud fija

Si se utiliza un ***código de longitud fija*****,** entonces todos los códigos de los distintos caracteres tendrán la misma longitud.

Los esquemas más usados son:

ASCII: Código Estándar Americano para Intercambio de Información

EBCDIC: Código Extendido de Intercambio Decimal Codificado en Binario

UNICODE: (desde 1991) : identificador numérico único para cada carácter o símbolo. Incluye todos los caracteres de uso comun.

Codificación
Códigos de longitud variable

- Es posible reducir significativamente el número de bits requeridos para representar los mensajes fuente si se emplea un *código de longitud variable.*

- En los códigos de longitud variable el número de bits requeridos puede variar de carácter en carácter.

- El objetivo es codificar los caracteres que aparecen más frecuentemente utilizando cadenas de bits más cortas, y para los caracteres que aparecen menos frecuentemente cadenas más largas.

Codificación
Códigos de prefijos

Código de prefijos (o código libre de prefijos): ningún código aparece como parte inicial de cualquier otro código, entonces la codificación es unívocamente decodificable.

Ejemplo: codificación de 6 caracteres: **a b c d e f**

	Código 1	***Código 2***	***Código 3***
Carácter	**Código de Longitud fija**	**Código de Longitud variable**	***Código de Prefijo***
a	000	00	11
b	001	111	1000
c	010	0011	1001
d	011	01	1010
e	100	0	0
f	110	1	1011

Codificación

El diseño de un código de longitud variable necesita cumplir con los principios:

- Asignar los códigos mas cortos a los símbolos mas frecuentes.
- Cumplir la propiedad de prefijo.
- Producir un código con la menor longitud promedio.

Los métodos mas conocidos de generar codificación variable son:

- Método de Shannon-Fano
- Codificación de Huffman

Codificación de Cadenas

- Problema de optimización: ***CODIFICAR una cadena de caracteres con un número mínimo de bits***: Dado un conjunto de caracteres y sus probabilidades, encontrar una codificación con la propiedad del prefijo tal que la longitud promedio de código por carácter sea mínima.

- El método general para encontrar un código de longitud variable con propiedad de prefijo, fue descubierto por David Albert Huffman en 1952 y se denomina “código de Huffman”.

- Los Códigos de Huffman son códigos de longitud variable, ampliamente usados y muy efectivos para la compresión de datos. Los más usados actualmente son códigos de Huffman adaptativos.

Algoritmo de Huffman


David Huffman

Algoritmo de Huffman: técnica para encontrar **códigos óptimos con la propiedad de prefijos** que aplica la **técnica greedy**.(*)

Se basa en la idea de construir ***árboles binarios*** para proporcionar una cadena de bits de longitud mínima para cualquier mensaje.

Los arboles que se van construyendo son *estrictamente binarios* y tienen almacenados los caracteres a codificar en sus hojas.

Se puede demostrar que el algoritmo de Huffman produce un código libre de prefijo y optimo. Aunque optimo el código puede no ser único.

(*) D.A. Huffman, "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., sept 1952, pp 1098-1102

Algoritmo de Huffman

Ideas del algoritmo de Huffman:

- Los candidatos son ***n*** símbolos del alfabeto a codificar, con la frecuencia de cada símbolo.
- Con cada uno de los candidatos se crea un árbol hoja, con el carácter almacenado y una frecuencia asociada al árbol.
- En cada paso se seleccionan los dos árboles que tengan menor frecuencia y se construye una árbol binario con ambos como hijos izquierdo y derecho y con frecuencia igual a la suma de los valores de sus hijos.
- Al final queda armado un árbol binario con los códigos de cada carácter.
- Los códigos son caminos en ese árbol y los caracteres están en las hojas.

Algoritmo de Huffman
Ejemplo

El tamaño del mensaje es de 10000 caracteres.

Los caracteres que aparecen en la cadena son:

- **A** aparece 3500 veces,
- **B** aparece 1000 veces,
- **C** aparece 2000 veces,
- **D** aparece 2000 veces,
- **-** aparece 1500 veces

Candidatos y frecuencias:

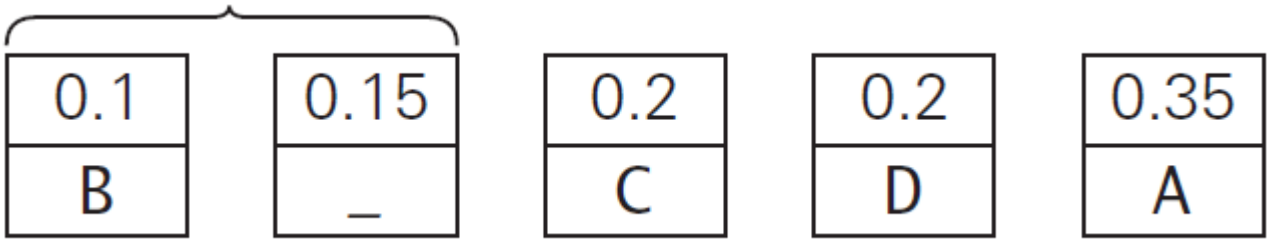
Símbolo	A	B	C	D	-
Frecuencia	0.35	0.10	0.20	0.20	0.15

Algoritmo de Huffman
Ejemplo

Datos:

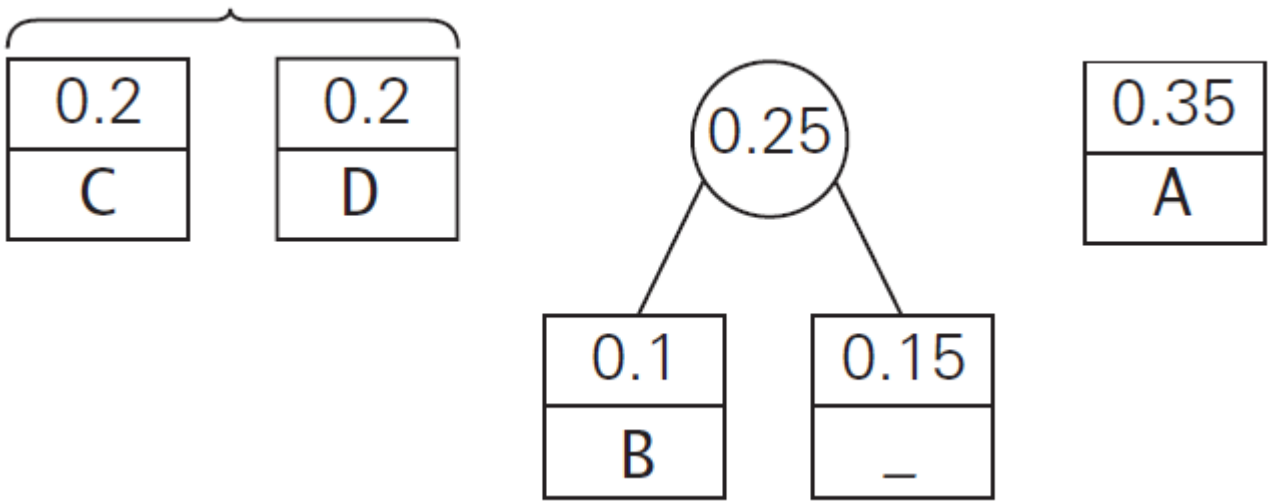
Símbolo	A	B	C	D	-
Frecuencia	0.35	0.10	0.20	0.20	0.15

Estado Inicial:



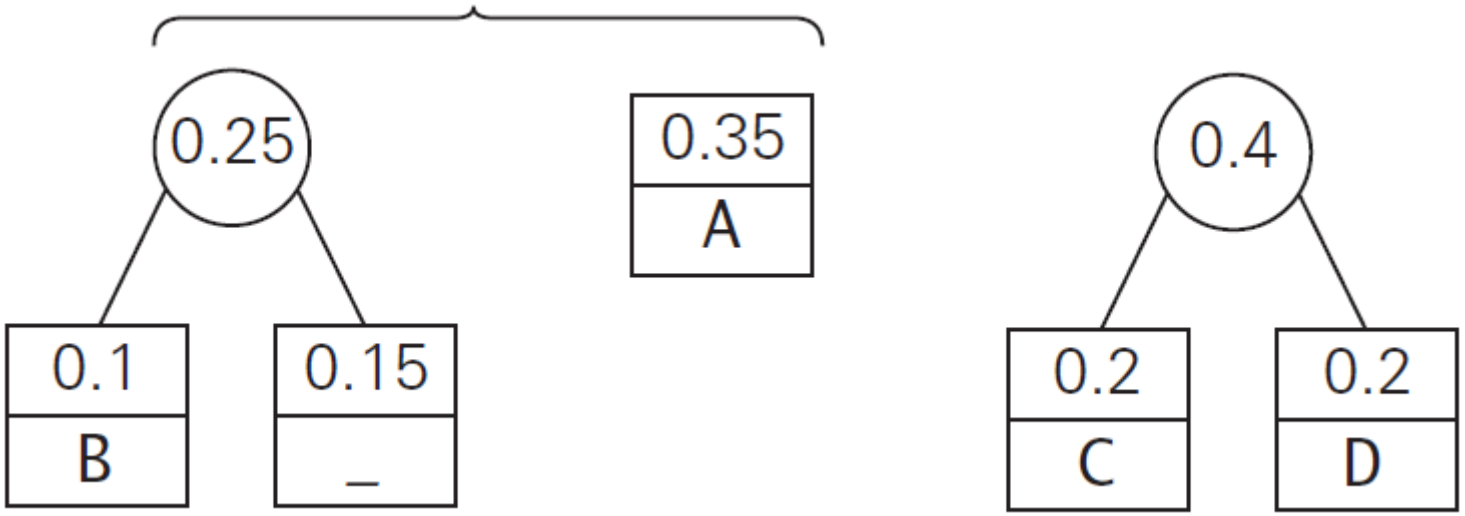
Algoritmo de Huffman
Ejemplo

Después del paso 1:



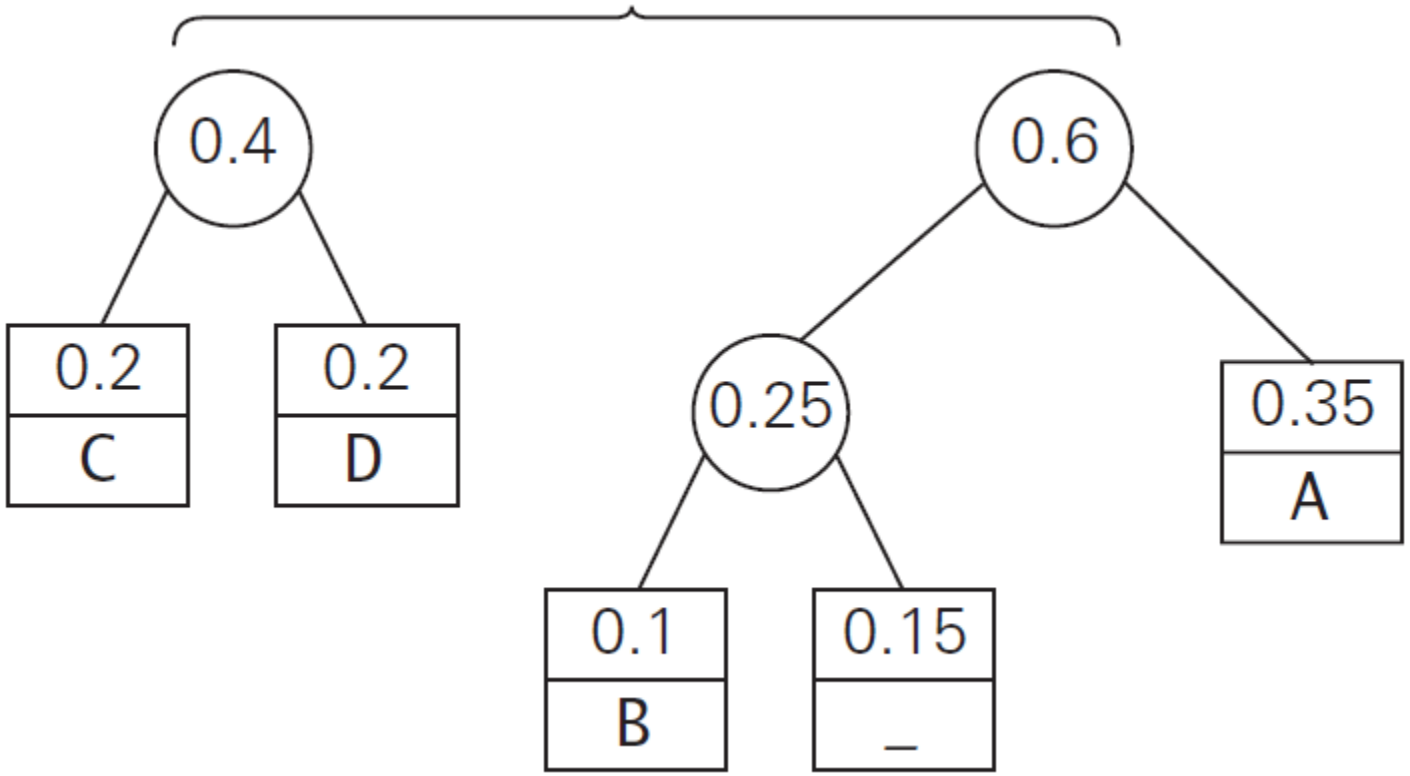
Algoritmo de Huffman
Ejemplo

Después del paso 2:



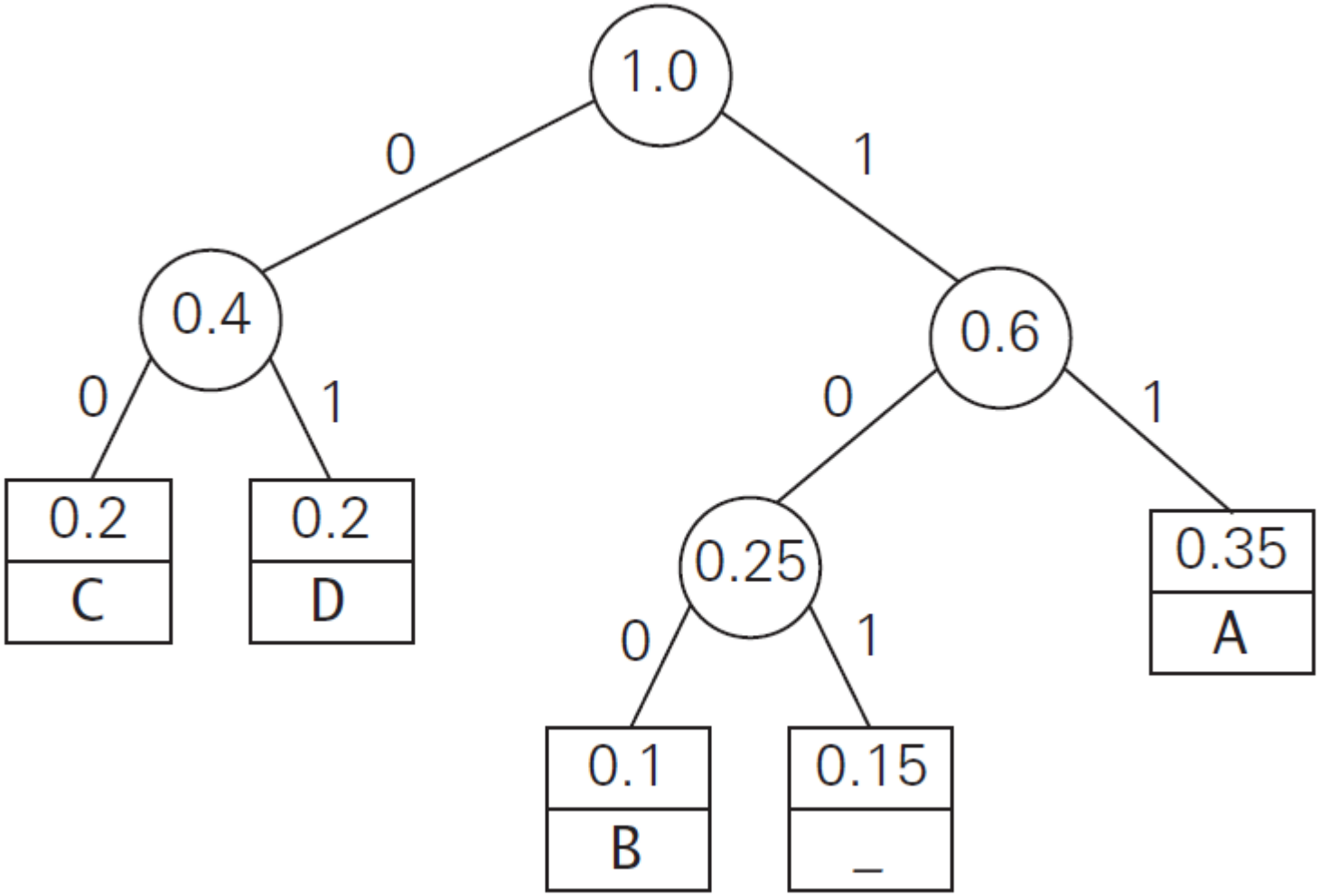
Algoritmo de Huffman
Ejemplo

Después del paso 3:



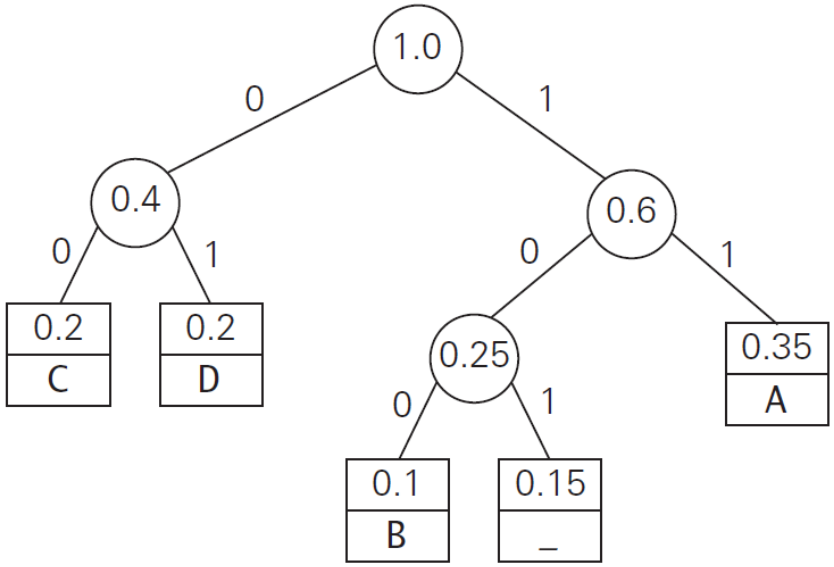
Algoritmo de Huffman
Ejemplo

Después del paso 4:



Algoritmo de Huffman
Ejemplo

El árbol de Huffman resultante**:**



Código de cada letra:

A → 11

B → 100

C → 00

D → 01

- → 101

Si se codifica el mensaje de 10000 caracteres con el Código de Huffman obtenido se necesitan:

3500*2 + 1000*3 + 2000*2 + 2000*2 + 1500*3 = **26500 bits.**

Con códigos de longitud fija de 3 bits se necesitan 30000 bits.

Algoritmo de Huffman

Con las frecuencias dadas y las codificaciones obtenidas, el promedio del número de bits por símbolo es:

$$2*0.35 + 3*0.1 + 2*0.2 + 2*0.2 + 3*0.15 = 2.25$$

En el ejemplo presentado en que se necesitan 3 bits para el código fijo, el código de Huffman una tasa de compresión del 25 %.

$$(3- 2.25)/3 * 100\%= 25\%.$$

En los experimentos realizados los códigos de Huffman han mostrado una tasa de compresión entre le 20% y el 80%, dependiendo de las características del texto que se comprime.

Resulta así el método de compresión más importante en la compresión de archivos, siempre que se conozca la frecuencia de ocurrencia de los símbolos.

Algoritmo de Huffman

La demostración de que el Algoritmo de Huffman encuentra siempre la solución optima se puede hacer a partir de 2 lemas:

Sea C un conjunto de caracteres y para cada carácter $c \in C$ se asigna su frecuencia $f(c)$. Sean x e y los dos caracteres del conjunto C que tienen las menores frecuencias.

Lema 1: Entonces existe una codificación de prefijo óptimo para el conjunto C en el cual los códigos de x y de y tienen la misma longitud y difieren solamente en el último bit.

Lema 2: Sea el conjunto de caracteres $C' = C - \{x,y\} \cup \{z\}$, donde cada carácter tiene asignada la misma frecuencia $f(c)$ que en C, excepto $f(z) = f(x) + f(y)$. Sea T' un árbol binario que representa los códigos binarios óptimos para el conjunto C'. Entonces el árbol T, obtenido de T' reemplazando el nodo hoja z por un nodo interno con los dos caracteres x e y como hijos, representa los códigos de prefijo óptimos para el conjunto C.

Algoritmo de Huffman

- Se puede mejorar este método básico usando una ***codificación dinámica* o *adaptativa*** de Huffman, en el cual el árbol se actualiza cada vez que se lee un nuevo símbolo en el código fuente.

- Es posible crear códigos de Huffman ternarios, cuaternarios, y, en general, n-arios. Para ello se realizan algunas modificaciones al algoritmo original.

- También hay versiones más sofisticadas como los algoritmos de ***Lempel-Ziv*** (Principalmente son dos: LZ77 y LZ78) que en lugar de tomar símbolos individuales trabaja con cadenas de símbolos.-