Analisis y diseño de algoritmos. Algoritmos voraces (Greedy)

Juan Gutiérrez

September 2019

Son algoritmos que construyen una solución escogiendo de manera local la mejor opción.

Son fáciles de diseñar, pero lo dificil es demostrar que el algoritmo devuelve la solución óptima en el largo plazo.

1 Intervalos disjuntos (sin pesos)

Sean $[s_1, f_1]$, $[s_2, f_2]$, $\ldots$, $[s_n, f_n]$ una secuencia de intervalos cerrados en la recta. Dos intervalos son *compatibles* si no se traslapan.

Problema Max-Intervalos-Disjuntos. Dada una secuencia de intervalos cerrados en la recta, encontrar un subconjunto de intervalos compatibles dos a dos de tamaño máximo.

Algunos posibles enfoques voraces para resolver el problema:

- Seleccionar el intervalo compatible que empieza antes (menor s_i)

  No funciona si el intervalo es muy grande:

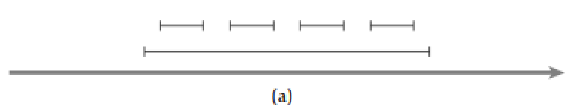


Figure 1: Tomada del libro Kleinberg, Algorithm Design

- Seleccionar el intervalo compatible con menor tamaño (menor $t_i - s_i$)

  No funciona en algunos casos:

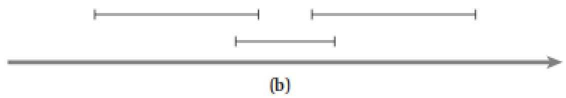


Figure 2: Tomada del libro Kleinberg, Algorithm Design

- Seleccionar el intervalo compatible con menor cantidad de intersecciones

  Es más dificil encontrar un contraejemplo, pero tampoco funciona siempre:

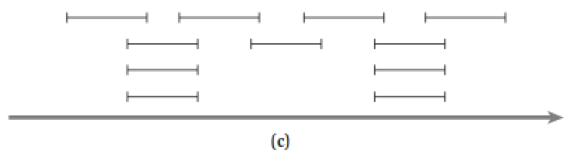


Figure 3: Tomada del libro Kleinberg, Algorithm Design

Una idea que **sí** funciona: tomar el intervalo compatible **con menor valor de su punta final**.

Recibe: un conjunto $\mathcal{I} = \{[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]\}$ de intervalos, ordenados de manera creciente por punta final

Devuelve: un subconjunto de intervalos compatibles dos a dos

Min-Intervalos-Disj($\mathcal{I}$)

$A = \emptyset$
while $\mathcal{I} \neq \emptyset$
    Sea $[s_i, f_i] \in \mathcal{I}$ tal que f_i es mínimo
    $A = A \cup \{[s_i, f_i]\}$
    $\mathcal{I} = \mathcal{I} \setminus \{[s_k, f_k] : [s_k, f_k] \cap [s_i, f_i] \neq \emptyset\}$
return A

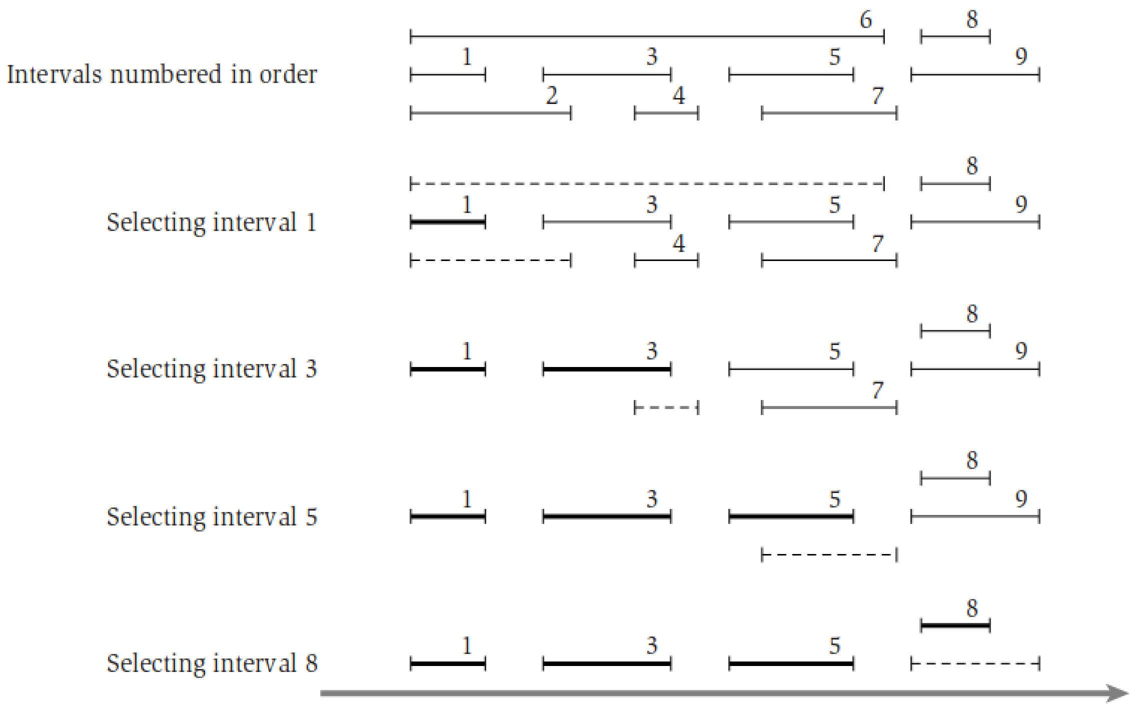


Figure 4: Tomada del libro Kleinberg, Algorithm Design

A continuación mostraremos que A es una solución óptima. Note que el problema puede tener muchas soluciones óptimas. Sea X una solución óptima cualquiera para el problema, basta mostrar que $|A| = |X|$.

Sean $i_1, i_2, \ldots, i_k$ los correspondientes índices de los intervalos en A en el orden en que fueron adicionados (línea 4). Sean $j_1, j_2, \ldots j_m$ los correspondientes índices de los intervalos en X. Debemos mostrar que $k = m$.

Propiedad 1.1. *Para todo índice $r \leq k$, se cumple $f_{i_r} \leq f_{j_r}$*

Proof. Por inducción en r. Si $r = 1$ entonces la propiedad se cumple debido a la elección hecha en la línea 3 del algoritmo.

Suponga ahora que $r > 1$. Por hipótesis de inducción, tenemos que $f_{i_{r-1}} \leq f_{j_{r-1}}$. Como X tiene intervalos compatibles, se cumple que $f_{j_{r-1}} < s_{j_r}$. Portanto $[s_{j_r}, f_{j_r}]$ está en el conjunto $\mathcal{I}$ al momento de elegir el intervalo i_r en la línea 3 del algoritmo. Debido a la manera que se elige, tenemos que $f_{i_r} \leq f_{j_r}$. □

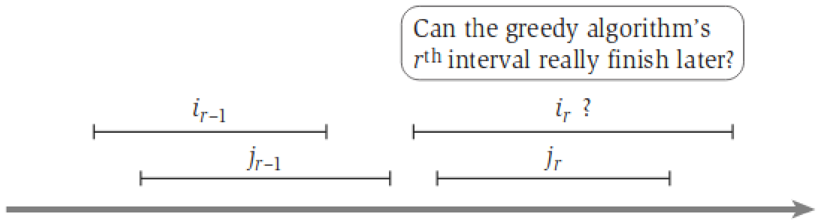


Figure 4.3 The inductive step in the proof that the greedy algorithm stays ahead.

Figure 5: Tomada del libro Kleinberg, Algorithm Design

Teorema 1.1. *El algoritmo* MIN-INTERVALOS-DISJ *hace lo pedido*

Proof. Suponga por contradicción que el conjunto A devuelto por el algoritmo no es una solución óptima al problema. Sea X una solución óptima. Por la Propiedad 1.1 (adoptando la notación necesaria), tenemos (cuando $r = k$) que $f_{i_k} \leq f_{j_k}$.

Como A no es óptima, entonces $k < m$ y, en la línea 5 de la iteración en donde se elije i_k, existe un intervalo $[s_{j_{k+1}}, f_{j_{k+1}}]$ en $\mathcal{I}$. Portanto el último índice a elegirse en A no es i_k, una contradicción.

$\square$

2 Planificación de tareas

Considere la situación en que una máquina debe atender n tareas, de manera contigua, cada una de las cuales cuenta con un tiempo de duración t_i y una fecha de entrega (deadline) d_i.

Si una tarea acaba a tiempo no tiene penalidad, si no acaba a tiempo tendrá una penalidad igual al tiempo que se retrasó respecto del deadline.

Más formalmente, si la tarea i es puesta en el intervalo $[s_i, f_i]$, la penalidad es cero si $f_i \leq d_i$ y es igual a $f_i - d_i$ en caso contrário.

Decimos que la *tardanza* de una asignación es igual al máximo de las penalidades de sus tareas.

Problema Min-Retraso-Tareas. Dada una secuencia de tareas a ser procesadas en una máquina, encontrar una secuencia de asignación a dichas tareas que minimiza la tardanza.

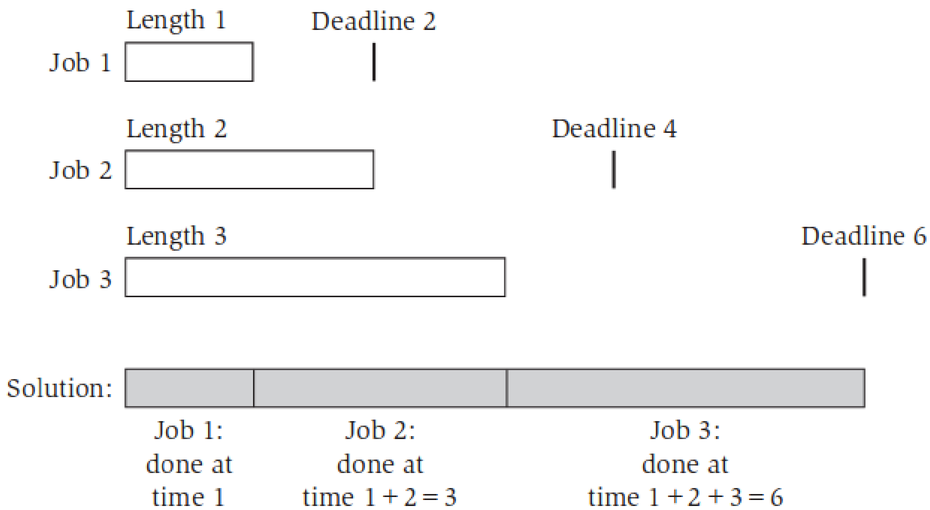


Figure 6: Tomada del libro Kleinberg, Algorithm Design

Algunos posibles enfoques voraces para resolver el problema:

- Seleccionar trabajo que dura menos (menor t_i)

  No funciona en este caso: $t_1 = 1$, $d_1 = 100, t_2 = 10, d_2 = 10$.

- Seleccionar trabajo con menor holgura (menor $d_i - t_i$)

  No funciona en este caso: $t_1 = 1$, $d_1 = 2, t_2 = 10, d_2 = 10$.

Una idea que **sí** funciona: **seleccionar trabajo con menor** d_i.

Recibe: Dos arreglos $[t_1, t_2, \ldots, t_n]$, $[d_1, d_2, \ldots, d_n]$, que guardan tiempos y deadlines de n trabajos, tal que $d_1 < d_2 < \cdots < d_n$.
Devuelve: Una asignación con menor retraso posible
MIN-RETRASO-GREEDY($\mathcal{I}$)
1: **return** $[1, 2, 3, \ldots, n]$

Mostraremos que el algortimo MIN-RETRASO-GREEDY resuelve de manera óptima el problema.

Teorema 2.1. MIN-RETRASO-GREEDY *resuelve de manera óptima el problema.*

Proof. Dado un arreglo W, el par (i, j) es una *inversión* en W si $i < j$ pero $W[i] > W[j]$. Sea X una solución óptima *con el menor número de inversiones.*

Mostraremos que X no tiene inversiones. Suponga por contradicción que X tiene por lo menos una inversión.

Tome una inversión (i, j) en X tal que $j - i$ es mínimo. Mostraremos que $j - i = 1$. Suponga por contradicción que $j - i > 1$. Entonces existe un índice k tal que $i < k < j$.

Si $X[i] > X[k]$ entonces (i, k) es una inversión, con $k - i < j - i$, una contradicción a la elección de (i, j). Si $X[i] < X[k]$, entonces, como $X[i] > X[j]$, tenemos que $X[j] < X[k]$ y (k, j) es una inversión con $j - k < j - i$, lo cual es nuevamente una contradicción a la elección de (i, j).

Del párrafo anterior, sea (i, j) una inversión en X con $j = i + 1$.

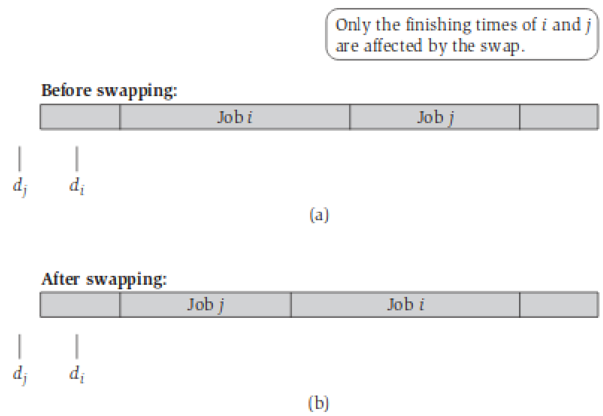


Figure 7: Tomada del libro Kleinberg, Algorithm Design

Sea X' la solución que resulta de intercambiar los valores de i y j en X, es decir, $X'[i] = X[j]$, $X'[j] = X[i]$ y $X'[k] = X[k]$ para todo $k \neq i, j$. Mostraremos que la tardanza en X' es menor o igual que la tardanza en X.

Como $j = i + 1$, para cualquier trabajo $k \neq i, j$ su tiempo final en X' no varía respecto de X, portanto la penalidad de dichos trabajos se mantienen en la solución X'. Luego, solo nos interesa analizar los trabajos i y j.

Sean $p(i), p(j), p'(i), p'(j)$ las penalidades de i, j en X y X' respectivamente.

Como $X[i] > X[j]$, tenemos que $d_i > d_j$. Luego,

$$p'(i) = d_j + p(j) - d_i < p(j)$$

$$p'(j) = d_j + p(j) - t_i - d_j = p(j) - t_i < p(j)$$

Y como $p(i) < p(j)$, tenemos que $p'(i), p'(j) < \max\{p(i), p(j)\}$. Eso implica que la tardanza en X' es menor o igual que la tardanza en X. Portanto, como X es óptima, X' también debe ser óptima. Pero X' tiene menos inversiones que X, una contradicción a la elección de X. $\square$

3 Una técnica general para demostraciones

Si queremos demostrar que nuestro algoritmo está correcto basta demostrar dos cosas.

1. *Elección voraz:* debemos demostrar que siempre existe una solución óptima que contiene a la eleccción voraz
2. *Subestructura óptima*: debemos demostrar que la subsolución dejada es óptima para el subproblema dejado por la elección voraz

Si se demuestran esos dos puntos, demuestro que mi voraz está correcto. Volvamos al problema de intervalos disjuntos para demostrar usando esta técnica.

3.1 Demostración para intervalos disjuntos

Recordemos el problema de intervalos disjuntos.

Sean $[s_1, f_1]$, $[s_2, f_2]$, $\cdots$, $[s_n, f_n]$ una secuencia de intervalos cerrados en la recta. Dos intervalos son *compatibles* si no se traslapan.

Problema Max-Intervalos-Disjuntos. Dada una secuencia de intervalos cerrados en la recta, encontrar un subconjunto de intervalos compatibles dos a dos.

Elección voraz: elegir el intervalo con menor f_i.

Planteamos el algoritmo recursivo:

Recibe: un conjunto $\mathcal{I} = \{[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]\}$ de intervalos, ordenados de manera creciente por punta final

Devuelve: un subconjunto de intervalos compatibles dos a dos

Min-Intervalos-Disj-Rec($\mathcal{I}$)

```
if I = ∅
    return ∅
I' = I \ {[s_i, f_i] : s_i ≤ f_1}
return {[s_1, f_1]} ∪ Min-Intervalos-Disj-Rec(I')
```

Lema 3.1 (Elección voraz)**.** *Existe una solución óptima para el problema que contiene el intervalo* $[s_1, f_1]$.

Proof. Sea X una solución óptima para el problema. Si X contiene a $[s_1, f_1]$ entonces no tenemos nada que probar.

Suponga entonces que $[s_1, f_1] \notin X$. Sea $[s_j, f_j]$ el intervalo en X con menor valor de f_j. Sea $X' = X \setminus \{[s_j, f_j]\} \cup \{[s_1, f_1]\}$. Mostraremos que X' es una solución para el problema. Para esto basta mostrar que $[s_1, f_1]$ es compatible con cualquier intervalo en $X \setminus \{[s_j, f_j]\}$.

Sea $[s_k, f_k]$ un intervalo cualquiera en $X \setminus \{[s_j, f_j]\}$. Note que

$$f_1 \leq f_j < s_k,$$

portanto $[s_1, f_1]$ es compatible con $[s_k, f_k]$.

Como X' es una solución para el problema y $|X| = |X'|$, concluimos que X' es una solución óptima al problema que contiene $[s_1, f_1]$. □

Lema 3.2 (Subestructura óptima)**.** *Si* X *es una solución óptima al problema que contiene a* $[s_1, t_1]$ *entonces* $X \setminus \{[s_1, t_1]\}$ *es una solución óptima al subproblema dejado por la elección voraz.*

Proof. Sea $\mathcal{I}$ la colección de intervalos del problema original. Sea $\mathcal{I}'$ la colección de intervalos luego de aplicar la elección voraz, es decir

$$\mathcal{I}' = \{[s_k, f_k] : f_1 < s_k\}.$$

Suponga por contradicción que $X' = X \setminus \{[s_1, t_1]\}$ no es una solución óptima para $\mathcal{I}'$. Entonces existe una solución Y' para $\mathcal{I}'$ con $|Y'| > |X'|$. Pero en ese caso $Y = Y' \cup \{[s_1, t_1]\}$ es una solución para $\mathcal{I}$ con tamaño $|Y| = |Y'| + 1 > |X'| + 1 = |X|$, contradicción. □

Ejercicio 3.1. *Describa un algoritmo eficiente que, dado un conjunto* $\{a_1, a_2, \ldots, a_n\}$ *de puntos en la recta, determine un conjunto mínimo de intervalos de tamaño 1 que contiene a todos los puntos. Justifique que su algoritmo es correcto usando la propiedades de elección voraz y subestructura óptima.*

4 Mochila fraccionaria

Recordemos el problema de la mochila

Problema Mochila-entera. Dado un conjunto $\{1, 2, \ldots, n\}$ de items cada uno con un peso natural w_i, un valor natural v_i y un número natural W, encontrar un subconjunto de items cuya suma de valores es la mayor posible, pero menor o igual a W.

El problema fraccionario permite elegir "fracciones de items" en cada elección.

Problema Mochila-fraccionaria. Dado un conjunto $\{1, 2, \ldots, n\}$ de items cada uno con un peso natural w_i, un valor natural v_i y un número natural W, encontrar un vector de racionales entre 0 y 1 $(x_1, x_2, \ldots, x_n)$ que maximize $\sum_{i=1}^{n} x_i v_i$ sobre la restricción $\sum_{i=1}^{n} x_i w_i \leq W$

Ejemplo: suponga $W = 50, n = 5$, $w = [40, 30, 20, 10, 20]$, $v = [840, 600, 400, 100, 300]$. Entonces $x = [1, 1/3, 0, 0, 0]$ es una solución viable para el problema, ya que $1 \cdot 40 + 1/3 \cdot 30 + 0 \cdot 20 + 0 \cdot 10 + 0 \cdot 20 = 50 \leq 50$.

Elección voraz: escoger siempre los items con mayor ratio valor/peso. Podemos suponer que $v_1/w_1 \leq v_2/w_2 \leq \cdots \leq v_n/w_n$. Tenemos el siguiente algoritmo.

Recibe: Una instancia v, w, W del problema Mochila-Fraccionaria
Devuelve: Una solución óptima para dicha instancia
MochilaFraccionaria-Greedy(v, w, W)

for $j = n$ **to** 1
  if $w[j] \leq W$
    $x_j = 1$
    $W = W - w[j]$
  else
    $x_j = W/w[j]$
    $W = 0$

8: **return** x

Note que dicho algoritmo no resuele el caso de mochila entera, en ese caso se debe aplicar programación dinámica (ver clases anteriores).

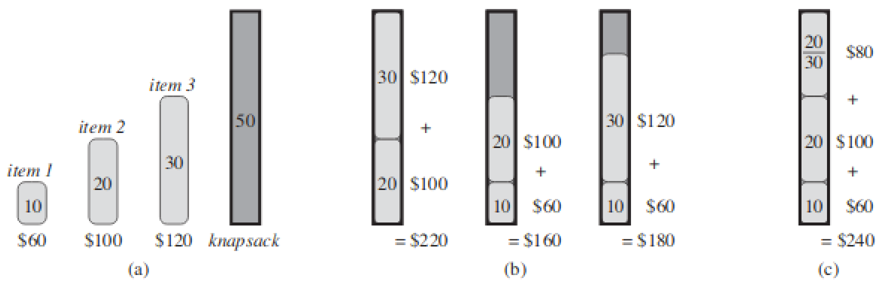


Figure 8: Tomada del libro Cormen, Introduction to Algorithms

A continuación demostraremos que nuestro algoritmo está correcto.

Lema 4.1 (Elección voraz)**.** *Existe una solución óptima* $(x_1, x_2, \ldots, x_n)$ *al problema tal que* $x_n = \min\{1, W/w_n\}$

Proof. Sea $\alpha = \min\{1, W/w_n\}$. Sea $(y_1, y_2, \ldots, y_n)$ una solución óptima al problema. Si $y_n = \alpha$ entonces no hay nada que probar. Suponga entonces que $y_n \neq \alpha$. Como y es una solución óptima, se cumple que $y \cdot w \leq W$. Portanto existe un $i \in \{0, 1, \ldots, n-1\}$ tal que $y_i > 0$.

Sea

$$\delta = \min\{y[i], (\alpha - y[n])\frac{w[n]}{w[i]}\}$$

y

$$\beta = \delta\frac{w[i]}{w[n]}$$

Defina x según

$$x[j] = \begin{cases} y[j] & \text{si } j \notin \{i, n\} \\ y[i] - \delta & \text{si } j = i \\ y[n] + \beta & \text{si } j = n \end{cases}$$

Note que

$$x \cdot w = y \cdot w - \delta w[i] + \beta w[n] = y \cdot w - \delta w[i] + \delta w[i] = y \cdot w.$$

También,

$$\begin{aligned}
x \cdot v &= y \cdot v - \delta v[i] + \beta v[n] \\
&= y \cdot v - \delta v[i] + \delta \frac{w[i]}{w[n]} v[n] \\
&= y \cdot v + \delta \left(\frac{w[i]}{w[n]} v[n] - v[i] \right) \\
&= y \cdot v + \delta w[i] \left(\frac{v[n]}{w[n]} - \frac{v[i]}{w[i]} \right) \\
&\geq y \cdot v
\end{aligned}$$

Concluimos que $xv \geq yv$, y como y es óptima, x también es óptima. □

Lema 4.2 (Subestructura óptima). *Si $(x_1, x_2, \ldots, x_n)$ es una solución óptima al problema con $x_n = \min\{1, W/w_i\}$, entonces $(x_1, x_2, \ldots, x_{n-1})$ es una solución óptima al subproblema dejado con $W = W - x_n w_n$.*

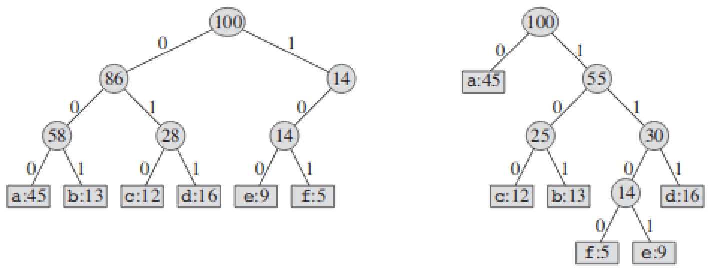
Proof. Suponga por contradicción que no es el caso. Sea $(y_1, y_2, \ldots, y_{n-1})$ una solución óptima al subproblema con peso máximo $W - x_n w_n$ que escoge los $n-1$ primeros items. Entonces $(y_1, y_2, \ldots, y_{n-1}, x_n)$ es una solución viable al problema original con valor mayor que $x \cdot v$, contradicción. □

5 Código de Huffman (Huffman codes)

Es una codificación de caracteres que permiten compactar archivos de texto. Es decir, transformar un archivo de caracteres en secuencia de bits.

Idea: usar pocos bits para los caracteres más frecuentes, y más bits para los más raros.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100



(a) (b)

Figure 9: Tomada del libro Cormen, Introduction to Algorithms

Códigos binarios de caracteres

Dado un conjunto C de caracteres, una *tabla de códigos* para C es una biyección entre C y algún conjunto de secuencias de bits. A la secuencia que corresponde a un caracter, le llamamos *código del caracter*.

Una tabla de códigos es *libre de prefijos* (prefix-free) si para cualquier par de caracteres x e y, el código de x no es prefijo del código de y.

El ejemplo de la figura anterior es libre de prefijos. En dicho ejemplo, la cadena $abacafe$ es codificada por 01010100011001101.

Codificación de archivos

Un *archivo* es una secuencia de caracteres. El conjunto de caracteres es el *alfabeto* del archivo.

El *peso* de un caracter en el archivo es la frecuencia (número de apariciones) de c, denotado por $p(c)$. Note que el número de caracteres del archivo es igual a $\sum_{c \in C} p(c)$.

Problema 5.1. *(Problema de compresión) Dado un archivo de caracteres, encontrar una tabla de códigos libre de prefijos que produzca un archivo codificado de tamaño mínimo.*

Un árbol de códigos para un conjunto C de caracteres es un arbol binario en que cada hoja corresponde a un elemento de C y cada nodo interno tiene exactamente dos hijos.

Sea $d(c)$ la profundidad del caracter c. Entonces el número total de bits usados en la codificación es

$$\sum_{c \in C} d(c)p(c).$$

Portanto, el problema anterior es equivalente a encontrar un árbol de códigos cuya suma de pesos por profundidad sea lo menor posible.

Árboles de Huffman

Sea $S = \{1, 2, \ldots, n\}$. Un *árbol de Huffman* respecto a S es cualquier colección Π de subconjuntos de S que cumple las siguientes propiedades.

1. para cada X y cada Y en Π, se tiene que $X \cap Y = \emptyset$, o $X \subseteq Y$ o $Y \subseteq X$,
2. $S \in \Pi$,
3. $\{\} \notin \Pi$,
4. todo elemento no minimal en Π, es la unión de otros dos elementos en Π.

Los elementos de Π son llamados *nodos*. El nodo S es llamado *raíz*. Los nodos minimales son llamados *hojas*. Todos los otros nodos son llamados *internos*.

Ejemplo: Sea $S = \{1, 2, 3, 4, 5, 6\}$, y $\Pi = \{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{2, 3\}, \{5, 6\}, \{4, 5, 6\}, \{2, 3, 4, 5, 6\}, \{1, 2, 3, 4, 5, 6\}\}$.

Si X, Y y $X \cup Y$ son nodos del árbol, decimos que X e Y son *hijos* de $X \cup Y$ y que $X \cup Y$ es el *padre* de X e Y. Un *ancestro* de X es cualquier nodo I tal que $X \subseteq I$. Si $I \neq X$ entonces I es un *ancestro propio*. En el ejemplo, $\{2, 3, 4, 5, 6\}$ es padre de $\{4, 5, 6\}$.

La *profundidad* de un nodo X es el número de ancestrales propios de X, será denotado por $d(x)$. En el ejemplo, $d(\{2, 3\}) = 2$.

Si X e Y son hojas, hijas del mismo padre, decimos que son hojas *hermanas*. Note que si X e Y son hojas hermanas, entonces $\Pi - \{X, Y\}$ también es un árbol de Huffman. Decimos que un árbol de Huffman tiene hojas *unitarias*, si cada una de sus hojas tiene solo un elemento.

Árboles de Huffman de peso mínimo

Una *ponderación* de un conjunto S es una atribución de peso numéricos a los elementos de S. Dada una ponderación p_i para cada $i \in S$, y $X \subseteq S$, denotamos por $p(X)$ a la suma $\sum_{i \in X} p_i$. Diremos que $p(X)$ es el *peso* de X.

Ejemplo: En el ejemplo anterior, podría asignar $p(1) = 45$, $p(2) = 13$, $p(3) = 12$, $p(4) = 16$, $p(5) = 9$, $p(6) = 5$.

El *peso* de un árbol de Huffman $p(\Pi)$, denotado por $p(\Pi)$ es la suma de pesos de sus nodos que no son raices, osea

$$p(\Pi) = \sum_{X \in \Pi - \{S\}} p(X).$$

En el ejemplo, $p(\Pi) = p(\{1\}) + p(\{2\}) + p(\{3\}) + p(\{4\}) + p(\{5\}) + p(\{6\}) + p(\{2,3\}) + p(\{5,6\}) + p(\{4,5,6\}) + p(\{2,3,4,5,6\}) = 45 + 13 + 12 + 16 + 9 + 5 + 25 + 14 + 25 + 55 = 219$.

Ejercicio 5.1. *Probar que* $p(\Pi) = \sum_{X \in \Gamma} p(X) d(X)$*, donde* Γ *es el conjunto de hojas de* Π.

Problema Min-Peso-Huffman. Dada una partición Γ y un conjunto S con una ponderación, encontrar un árbol de Huffman de peso mínimo de entre las que tienen como hojas a Γ.

En el ejemplo, $S = \{1,2,3,4,5,6\}, \Gamma = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}, p(1) = 45, p(2) = 13, p(3) = 12, p(4) = 16, p(5) = 9, p(6) = 5$.

Algoritmo de Huffman

Recibe: Un conjunto S, una ponderación p de S y una partición Γ de S
Devuelve: Un árbol de Huffman óptimo (con peso mínimo) que tiene a Γ como conjunto de hojas

Huffman(S, p, Γ)
if $|\Gamma| = 1$
  return Γ
Sea X un elemento en Γ con ponderación mínima
$\Gamma = \Gamma - X$
Sea Y un elemento en Γ con ponderación mínima
$\Gamma = \Gamma - \{Y\}$
$\Gamma = \Gamma \cup \{X \cup Y\}$
return $\{X, Y\} \cup$ Huffman(S, p, Γ)

Ejemplo: sea $S = \{1, 2, \ldots, 6\}$, $p(1) = 45, p(2) = 13, p(3) = 12, p(4) = 16, p(5) = 9, p(6) = 5$ y $\Gamma = \{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}\}$.

El algoritmo produce el árbol $\{\{1\},\{2\},\{3\},\{4\},\{5\},\{6\},\{6,5\},\{6,5,4\},\{2,3\},\{2,3,4,5,6\}, \{1,2,3,4,5,6\}\}$. Otra manera de representar el árbol es $(1,((3,2),((6,5),4)))$. Su peso es 224. Note que otra posible solución es $((1,2),((5,6),(3,4)))$, que tiene peso 242 y por lo tanto no es óptima.

Ejercicio 5.2. *Corra el algoritmo de Huffman para* $S = \{1,2,3,4,5,6\}$ *con hojas unitarias y ponderación* $p(1) = p(2) = p(3) = p(4) = p(5) = p(6)$*. De tambíen un árbol de Huffman no óptimo para esa misma ponderación.*

Implementación con fila de prioridades

Recibe: Un conjunto S, una ponderación p de S
Devuelve: Un árbol de Huffman óptimo (con peso mínimo) que tiene a los elementos de S como conjunto de hojas

Huffman-FilaPrioridades(S, p)
$n = |S|$
$Q =$ Iniciar-FP()

```
for i = 1 to n
    z.peso = p(i)
    z.left = NIL
    z.rigth = NIL
    INSERT-FP(Q, z)
for i = 1 to n − 1
    x = EXTRAERMIN-FP(Q)
    y = EXTRAERMIN-FP(Q)
    z.left = x
    z.rigth = y
    z.peso = x.peso + y.peso
    INSERT-FP(Q, z)
return  EXTRAERMIN-FP(Q)
```

El tiempo de ejecución es $O(n \lg n)$ si la fila de prioridades es implementada como heap.

Ejercicio 5.3. *Corra* HUFFMAN-FILA-PRIORIDADES *para* $S = \{1, 2, 3, 4, 5, 6\}$ *con hojas unitarias y ponderación* $p(1) = p(2) = p(3) = p(4) = p(5) = p(6)$.