Práctica 4. Generalización del método de Huffman

26 de octubre del 2022 **Marcos Hidalgo Baños**

a) Implementación del método de Huffman generalizado.

Nota Importante.

El algoritmo diseñado realiza correctamente el proceso de cifrado como se indica en las transparencias. Sin embargo, no he conseguido realizar la asignación del símbolo menor (el cero) al mensaje o grupo de mensajes con mayor probabilidad (tal y como se nos indicó en los ejercicios en clase).

El fallo se encuentra en la asignación del símbolo a escribir para cada mensaje, puesto que en esta implementación siempre se asigna al primero el símbolo menor (algo que no siempre es cierto).

Esto se ve en las líneas comentadas de la 33 a la 36, donde encuentro la mayor probabilidad para intentar asignarle a ella primero el símbolo menor, pero no soy capaz de seguir asignando al resto los sucesivos símbolos.

CONCLUSIÓN.

Una manera que seguramente funcione es emplear alguna estructura de datos como un mapa o un árbol, que nos permita recorrer todos los mensajes sin saltarnos ninguno y tratar con prioridad al de mayor probabilidad.

```
1  HuffmanGeneralizado <- function(mensajes, simbolos, probabilidades) {
2    # @param mensajes, vector de mensajes a codificar
3    # @param simbolos, vector con los simbolos del alfabeto
4    # @param probabilidades, vector de probabilidades de cada mensaje
5
6    # PASO 1.
7    if (sum(probabilidades) != 1) {
8      # Comprobamos que las probabilidades suman exactamente uno
9      return('ERROR. Las probabilidades no suman 1.')
10     # Si es el caso, debemos terminar inmediatamente
11   }
12   ordenado = OrdenarVector(probabilidades)
13   # Ordenamos las probabilidades descendentemente
14
15   # PASO 2.
16   n = length(mensajes) # numero de mensajes a codificar
17   D = length(simbolos) # total de simbolos para codificar
18   r = (n-2)%%(D-1) # resto de la division entera
```

```
20  # PASO 3.
21  codigo <- vector(mode = "character", length = n)
22  # Vector solucion (ahora vacio) con la codificacion asignada
23
24  i = n-(r+1) # Puntero a los ultimos mensajes a codificar
25  s = D-(r+1) # Puntero a los simbolos a escribir en esta iteracion
26  p_ac = 0    # Nueva probabilidad acumulada de los ultimos mensajes
27  m = 0
28
29  while (length(ordenado) != 1) {
30    # Acabaremos cuando el vector de prob. ordenado sea un solo valor
31    posiciones = c(i:n)
32
33    # if (length(mensajes) != length(ordenado) ) {
34    #   m = max(ordenado[posiciones])
35    #   s = ((n-m) %% D) + 1
36    # }
37
38    for (j in posiciones) {
39      # Para cada mensaje a partir de i hasta n
40
41      if (codigo[j] == "")
42        { # Si es la primera iteracion del asignado
43          codigo[j] = simbolos[s]
44          # Escribimos el simbolo correspondiente
45        }
46      else
47        { # Si no, pegamos delante el nuevo simbolo
48        for (k in n:length(codigo)) {
49          # Esto debe realizarse a todos los sucesivos simbolos
50          codigo[k] = paste(simbolos[s], codigo[k])
51          # Pegamos el nuevo simbolo delante del que ya estaba
52        }
53      }
54      print(codigo)
55
56      if (s < D) {
57        # Avanzamos al siguiente simbolo
58        s = s + 1
59      } else {
60        # O comenzamos desde el primer simbolo
61        s = ((s+1) %% D) + 1
62      }
63
64      p_ac = p_ac + ordenado[j]
65      # Acumulamos las prob. de los mensajes parcialmente codificados
66    }
```

```
    # PASO 4.
    ordenado = OrdenarVector(ReagruparVector(probabilidades, i, p_ac))
    # Recalculamos el nuevo vector ordenado a partir de la prob. acumulada
    print(ordenado)

    p_ac = 0  # Reseteo de la variable acumulativa de probabilidades
    n = length(ordenado)  # Actualizacion de la longitud tras el reagrupe
    i = n - (D-1) # Colocacion del indice sobre la siguiente
    s = 1 # Colocamos el puntero en el primer simbolo

  }

  return(codigo)
}
```

Pruebas realizadas con los ejercicios de la relación.

```
# Pruebas con el ejemplo hecho en clase de Huffman ternario
mensajes = c('a1','a2','a3','a4','a5','a6','a7','a8')
probabilidades = c(0.4,0.2,0.1,0.1,0.05,0.05,0.05,0.05)
simbolosTernario = c('0','1','2')
HuffmanGeneralizado(mensajes, simbolosTernario, probabilidades)
```

```
[1] ""  ""  ""  ""  ""  ""  "1" ""
[1] ""  ""  ""  ""  ""  ""  "1" "2"
[1] 0.40 0.20 0.10 0.10 0.10 0.05 0.05
[1] ""  ""  ""  ""  "0" ""  "1" "2"
[1] ""  ""  ""  ""  "0" "1" "1" "2"
[1] ""    ""    ""    ""    "0"   "1"   "2 1" "2 2"
[1] 0.4 0.2 0.2 0.1 0.1
[1] ""    ""    "0"   ""    "0"   "1"   "2 1" "2 2"
[1] ""    ""    "0"   "1"   "0"   "1"   "2 1" "2 2"
[1] ""      ""      "0"     "1"     "2 0"   "2 1"   "2 2 1" "2 2 2"
[1] 0.4 0.4 0.2
[1] "0"     ""      "0"     "1"     "2 0"   "2 1"   "2 2 1" "2 2 2"
[1] "0"     "1"     "0"     "1"     "2 0"   "2 1"   "2 2 1" "2 2 2"
[1] "0"       "1"       "2 0"     "2 1"     "2 2 0"   "2 2 1"   "2 2 2 1" "2 2 2 2"
[1] 1
[1] "0"       "1"       "2 0"     "2 1"     "2 2 0"   "2 2 1"   "2 2 2 1" "2 2 2 2"
```

La solución correcta realizada en clase debería ser (0, 2, 11, 12, 101, 102, 1001, 1002) Sin embargo, mi algoritmo devuelve (0, 1, 20, 21, 220, 221, 2221, 2222) puesto que siempre asigna 0 al primer mensaje en orden. Notar la similitud entre ambas y lo relativamente sencillo que sería adaptar mi algoritmo para que sea correcto.

Funciones auxiliares.

```r
OrdenarVector <- function(probabilidades) {
  # @param probabilidades, vector de probabilidades

  return(rev(sort(probabilidades)))
  # Ordena descendentemente el vector
}

ReagruparVector <- function(probabilidades, i, p_ac) {
  # @param probabilidades, vector original a reducir
  # @param i, indice que indica el tope superior

  new_vector = probabilidades[-c(i:length(probabilidades))]
  # Recortamos el resto de probabilidades que colapsan en una
  new_vector[i] = p_ac
  # La ultima posicion sera la acumulada de las sucesivas (que no estaran)
  return(new_vector)
}
```

➔ sort (x)

Función para ordenar ascendentemente los valores de un vector.

➔ rev (x)

Revierte el orden del vector pasado por parámetro.

Junto con sort, permite ordenar un vector en sentido descendente.

➔ Operador ‘%%’

Operador modulo de división entera para obtener el resto.

➔ paste (x, y)

Función para la concatenación de Strings.

Resulta en ‘xy’ (al primero se le concatena el segundo)

➔ vector (mode = “character”, length = n)

Crea un vector vacío con longitud predeterminada de tipo cadena de caracteres.

➔ length (x)

Obtiene la longitud del vector para cualquier tipo de vector de valores.