



ITESO
Universidad Jesuita
de Guadalajara

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE
Departamento de Electrónica, Informática y Sistemas

Tutorial sobre árboles binarios en Matlab/Octave con aplicación al algoritmo de Huffman

L. M. Bazdresch
`miguelbaz@iteso.mx`

Noviembre 2009

Copyright (c) 2009 L. Miguel Bazdresch, Universidad ITESO

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1. Introducción

Una gran variedad de algoritmos en ingeniería sugieren, de forma natural, representar los datos con la estructura de un árbol binario. Un ejemplo clásico es el algoritmo de Huffman, que comprime de forma óptima un archivo.

Con frecuencia ocurre también que un estudiante o profesional necesita implementar esta estructura de datos, sin haber llevado un curso formal, o bien en un lenguaje que no se asocia comúnmente con estructuras como ésta, por no contar con apuntadores a memoria.

En este tutorial se ofrece un punto de partida para implementar, de la forma más simple posible, un árbol binario en Matlab u Octave.

2. Motivación: el Algoritmo de Huffman

El algoritmo de Huffman genera un código para los símbolos de una fuente \mathcal{X} . El código generado por este algoritmo tiene la propiedad de ser el código instantáneo con longitud promedio más cercana posible a la entropía de \mathcal{X} . En este sentido el algoritmo es óptimo.

El algoritmo tiene como entrada un vector de probabilidades $\mathcal{P}_{\mathcal{X}}$ correspondientes a los símbolos de la fuente \mathcal{X} . La salida del algoritmo consiste en las palabras de código para cada símbolo.

A manera de ejemplo, digamos que $\mathcal{P}_{\mathcal{X}} = [0.25 \ 0.25 \ 0.2 \ 0.15 \ 0.15]$. El algoritmo une los símbolos como se muestra en la figura (1). Las palabras de código resultantes son 00, 01, 10, 110, y 111.

Este algoritmo es sencillo de correr en papel, para fuentes con pocos símbolos. Sin embargo, cualquier aplicación práctica requiere programar el algoritmo en una computadora. Esta tarea no es tan sencilla como pudiera parecer a primera vista. En este documento se ofrecen algunas ideas y código para realizar con más certidumbre una implementación en Octave/Matlab.

3. Estructuras de Datos y Árboles

Una estructura de datos es una forma de organizar y relacionar datos en la memoria de una computadora, de manera que puedan ser usados eficientemente. Generalmente, el problema mismo que se quiere resolver sugiere una forma de organizar los datos. Por ejemplo, para operaciones de álgebra lineal conviene usar vectores y matrices. Una tabla *hash* está diseñada para asociar identificadores (por ejemplo un nombre de una persona) con un valor (por

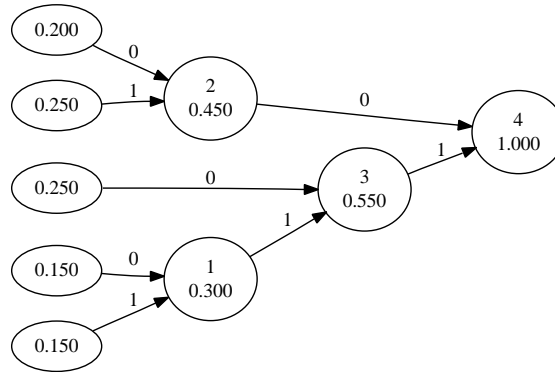


Figura 1: Ejecución del algoritmo de Huffman. Los círculos de la columna izquierda son las probabilidades iniciales. Los demás círculos tienen indicado, arriba, la iteración del algoritmo en que fueron insertados al árbol, y abajo, la probabilidad de cada símbolo compuesto.

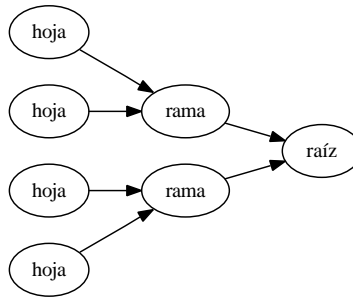


Figura 2: Relación entre nodos de un árbol binario.

ejemplo un número de teléfono). Un grafo puede representar una región, con ciudades en los vértices y carreteras en las aristas, para calcular distancias o las mejores rutas entre ellas.

La figura (1) sugiere una forma de organizar los datos que el algoritmo de Huffman va generando. Esta organización se conoce como *árbol binario*. Un árbol binario contiene tres tipos de nodos: hojas, ramas y raíces. Cada nodo contiene información, y está relacionado con otros nodos (ver figura (2)). Además, se dice que una raíz es *padre* de dos ramas, cada una de las cuales es padre de otras dos ramas o dos hojas. Cada rama y cada hoja es hija de otra rama o de una raíz. El árbol se llama binario precisamente porque cada nodo rama o raíz tiene exactamente dos hijos.

Es clara la similitud entre el diagrama obtenido ejecutando el algoritmo

de Huffman y un árbol binario. Esto nos sugiere que necesitamos implementar esta estructura de datos para implementar el algoritmo.

4. Árboles Binarios en Matlab y Octave

Una implementación de árboles binarios requiere lo siguiente:

- Definir los nodos y una forma de crearlos
- Definir una forma de almacenar los nodos
- Crear funciones auxiliares:
 - Guardar información en un nodo
 - Determinar si un nodo es raíz, rama u hoja
 - Insertar un nuevo nodo en un árbol

4.1. Definición de nodos

Podemos determinar qué información hay que guardar en un nodo a partir de la figura (1):

- Un identificador
- La probabilidad del símbolo asociado al nodo
- Apuntador al nodo padre
- Apuntador a un hijo a través de una arista 1
- Apuntador a un hijo a través de una arista 0

Un nodo puede ser definido con esta función:

```
%  
% bt_node  
%  
% node = bt_node(i);  
%  
% Regresa un nodo vacío  
  
function node = bt_node();  
    node.parent = -1; % Posición de nodo padre
```

```

        node.prob = 0;      % Probabilidad de este nodo
        node.uno = -1;      % Posición de nodo hacia uno
        node.cero = -1;     % Posición de nodo hacia cero
    end

```

Los apuntadores *parent*, *uno*, y *cero* son números que indican la posición en el arreglo (ver abajo) donde se encuentran los nodos padre y los dos hijos, respectivamente. El valor -1 se utiliza para indicar que no se apunta a nada.

4.2. Almacenamiento de los nodos

Los nodos pueden ser almacenados en un vector. El identificador del nodo es su posición en el vector. La siguiente función inserta un nuevo nodo vacío en un arreglo:

```

%
% [BTout i] = bt_insert( BTin )
%
% Inserta un nodo en el arreglo BTin
%
% BTin es un arreglo donde queremos insertar un nuevo nodo
% BTout es igual a BTin pero con un nuevo nodo
% i es la posición donde se insertó el nodo

function [BTout i] = bt_insert( BTin )
    i = length(BTin) + 1;
    BTout = [ BTin bt_node ];
end

```

Octave y Matlab siempre pasan valores a funciones por referencia, y no hay forma de hacer que una función modifique el valor de una variable fuera de la misma función. Por tanto, para modificar el vector de nodos, es necesario pasarle a la función el arreglo original y recibir de regreso (como salida de la función) el arreglo nuevo.

El arreglo con nodos es propiamente el árbol binario.

4.3. Guardar información en un nodo

La siguiente función permite cambiar los valores de un nodo:

```

%
% BTout = bt_set( BTin, i, parent, prob, up, down )

```

```

%
% Da valores a los campos de un nodo
%
% BTout es un arreglo con el nodo i actualizado
% BTin es el arreglo de nodos que se desea actualizar
% i es el número de nodo que se va a actualizar
% parent, prob, up, down son los nuevos valores del nodo i

function BTout = bt_set( BTin, i, parent, prob, uno, cero )
    node.parent = parent;
    node.prob = prob;
    node.uno = uno;
    node.cero = cero;
    BTin(i) = node;
    BTout = BTin;
end

```

4.4. Determinar si un nodo es raíz, rama u hoja

Estrictamente hablando, un árbol binario siempre tiene una sólo raíz y cada nodo tiene o dos hijos, o ninguno (en el caso de las hojas). Sin embargo, durante la construcción del árbol esta condición no se cumple, y por eso conviene definir una raíz como cualquier nodo sin padre, y una hoja como cualquier nodo sin hijos.

Así, estas funciones consisten simplemente en recorrer el árbol y examinar cada nodo para determinar si tiene padre o si tiene hijos. La codificación de estas funciones se deja como ejercicio para el lector.

4.5. Ejemplos

- Crear un árbol binario con un nodo vacío

```
BT = bt_insert([]);
```

- Añadir otros dos nodos al árbol y poner sus probabilidades en 0.45, 0.35 y 0.2.

```

BT = bt_insert(BT);
BT = bt_insert(BT);
BT = bt_set(BT,1,-1,.45,-1,-1);
BT = bt_set(BT,2,-1,.35,-1,-1);

```

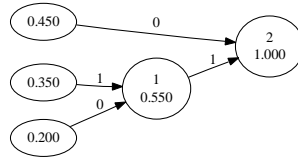


Figura 3: Árbol de ejemplo.

```
BT = bt_set(BT,3,-1,.2,-1,-1);
```

- Conectar los nodos 2 y 3 a un nuevo nodo y poner su probabilidad en 0.55.

```
BT = bt_insert(BT);
BT = bt_set(BT,2,4,BT(2).prob,-1,-1);
BT = bt_set(BT,3,4,BT(3).prob,-1,-1);
BT = bt_set(BT,4,-1,.55,2,3);
```

Aquí estamos diciendo que el nodo padre de los nodos 2 y 3 es el 4; a su vez, el nodo 4 tiene hijos 2 (con valor 1) y 3 (con valor 0).

- Para terminar el árbol, conectar los nodos 1 y 4 a un nuevo nodo.

```
BT = bt_insert(BT);
BT = bt_set(BT,1,5,BT(1).prob,-1,-1);
BT = bt_set(BT,4,5,BT(4).prob,BT(4).uno,BT(4).cero);
BT = bt_set(BT,5,-1,1,4,1);
```

Con esto terminamos el árbol.

Para verificar si el árbol está bien construido, podemos usar la función `bt_print(BT)` (que se encuentra en el mismo paquete al que pertenece este documento). Esta función requiere tener instalado el paquete **graphviz** con el *backend* llamado *cairo*¹. Corriendo esta función obtenemos el diagrama mostrado en la figura (3), donde podemos ver que en efecto el árbol está bien construido.

¹En Ubuntu, se pueden instalar ejecutando `sudo apt-get install graphviz graphviz-cairo`.

5. Implementación del Algoritmo de Huffman

La implementación del algoritmo consiste en realizar de manera automática lo que hicimos a mano en la subsección 4.5. Una función que nos hace falta es determinar los dos nodos raíz del árbol que tienen la menor probabilidad (recuerde que hemos definido un nodo raíz como un nodo que no tiene padre). Esto se puede realizar con la función presentada abajo, que regresa los índices de los dos nodos raíz con menor probabilidad. La implementación del resto del algoritmo se deja como ejercicio para el lector.

Vale la pena notar que el código que se ha presentado tiene como objetivo la mayor claridad de exposición y no el mejor rendimiento. Para trabajar con árboles de tamaño grande será necesario optimizar el código para obtener velocidades razonables de procesamiento.

```
%
% A = bt_sort2( BT )
%
% Encuentra los índices de los dos nodos raíz con menor probabilidad
%
% BT es un arreglo de nodos
% A es un arreglo (1x2) con los índices de los nodos buscados
% A(1) es el nodo con menor probabilidad
% A(2) es el nodo con segunda menor probabilidad
% Si BT tiene una sola raíz, el segundo elemento de A es -1
% Si BT no tiene raíces, ambos elementos de A son -1

function A = bt_sort2( BT )
    A = [-1 -1];
    r = bt_roots( BT );
    if( length(r) == 0 )
        return;
    else
        % formar un arreglo con las probabilidades de las raíces
        b = [];
        for i=1:length(r)
            b = [b BT(r(i)).prob];
        end
        % ordenar
        % i(1) es la menor probabilidad, i(2) la 2a menor
        [o i] = sort(b);
```

```
        if( length(r) == 1 )
            A = [r(i(1)) -1];
        else
            A = [r(i(1)) r(i(2))];
        end
    end
end
```