



(75.29 / 95.06) TEORÍA DE ALGORITMOS

TRABAJO PRÁCTICO N.º 2 Grupo: “Los Fieles de Cardozo”

2 de Octubre de 2019

Integrante	Padrón	Correo electrónico
Salvador, Yago	— # 99 725 —	yagosalvador@gmail.com
Nenadovit, Emmanuel Angel	— # 91 734 —	emmanuelnenadovit@gmail.com
Untrojb, Kevin	— # 97 866 —	oliverk12@hotmail.com
Bocchio, Guido	— # 101 819 —	guido@bocch.io

1. Resumen

En el presente informe se expone el análisis y resultados de cada consigna. La primera parte consiste en resolver un problema en el cual se usa teoría y algoritmos de grafos, se expone la solución análisis y algoritmo utilizado. La segunda parte es un acercamiento a los algoritmos *Greedy*. Se analiza el algoritmo de compresión de datos *Huffman* y finalmente se programa. La última parte es optativa y consiste en analizar un *paper* donde se debe fundamentar si el algoritmo propuesto por *Donald B. Johnson* es tractable.

2. Despliegue de ayuda eficiente

El enunciado presenta un problema donde se debe minimizar la cantidad de equipos a armar cubriendo todo los centros urbanos existentes. Se considera que cada ciudad es el vértice o nodo de un grafo dirigido, las aristas son las vías de comunicación que siguen funcionando. Como entrada se reciben las aristas dirigidas del grafo.

La idea general de la solución es encontrar todas componentes fuertemente conexas del grafo principal para poder separarlo en sub-grafos fuertemente conexas. Aplicado al problema dado, al encontrar las componentes fuertemente conexas, se asegura que se puede partir de una ciudad dentro del sub-conjunto y volver a la misma recorriendo varias ciudades, entonces es posible optimizar la cantidad de equipos enviados.

Para poder obtener las componentes fuertemente conexas se decidió utilizar el algoritmo de *Kosaraju*. Este algoritmo consiste en recorrer el grafo con *Deep First Search (DFS)* mientras se van numerando los nodos recorridos hasta llegar a un nodo que no tenga adyacentes sin recorrer. Se podría utilizar la estructura de datos *Stack* para poder almacenar los nodos sin adyacentes primero y seguido seguir recorriendo hasta el final. Posterior a este primer paso se genera el grafo transversal invirtiendo las aristas y finalmente se va desapilando del stack los nodos. Todos los nodos adyacentes son componentes fuertemente conexas.

Algoritmo 1 Algoritmo de *Kosaraju*

inicializar $c:=0$

mientras todos los nodos no estén marcados **hacer**

 Elegir un nodo u

 hacer DFS(u)

 incrementar c

 marcar x como c

Girar las aristas del grafo (generar el grafo transversal)

para nodo u marcado de n hasta 1 **hacer**

 agrupar todos los nodos accesibles desde u y guardarlos como un conjunto

devolver Todos los conjuntos

Analizando la complejidad temporal de este algoritmo, siendo e la cantidad de aristas y v la cantidad de nodos empezamos por la función DFS es $\mathcal{O}(e+v)$. Crear el grafo transversal tiene una complejidad de $\mathcal{O}(e+v)$ ya que también hay que recorrer el grafo completo. Finalmente la complejidad del ciclo *for* final es análoga a llamar a DFS del grafo transversal. También tiene complejidad $\mathcal{O}(e+v)$. De esta forma la complejidad

total del algoritmo es $\mathcal{O}(e + v)$. La complejidad temporal del algoritmo se puede decir que es óptima, ya que es necesario recorrer todos los vértices y aristas para poder encontrar las componentes fuertemente conexas, sería imposible hacerlo sin poder recorrer algún vértice o arista ya que se tendría incompleta la información del grafo. Si se analiza la complejidad espacial, se obtiene $\mathcal{O}(v)$ ya que en el peor caso, debiera guardar todos los vertices en el stack.

Un uso practico que se le podría asociar a la búsqueda de componentes fuertemente conexas en un grafo es en la búsqueda de grupos de personas conocidas dentro de una red social. Si se considera que cada perfil personal en una red social es un vértice y la relación (sea de amistad, conectado o seguidor) es una arista, las componentes fuertemente conexas en este caso seria cada grupo de personas que se conocen entre si y se podría establecer cierta relación real entre los integrantes de este grupo.

3. Comunicación satélital

Como parte del programa espacial argentino se pidió programar el algoritmo de compresión de datos *Huffman* para utilizar en la transmisión de las imágenes enviadas desde los satélites.

Un algoritmo *Greedy* es una estrategia de solución que consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Una vez llegada a la solución se demuestra que es óptima o que tiene cierta cercanía con un óptimo propuesto.

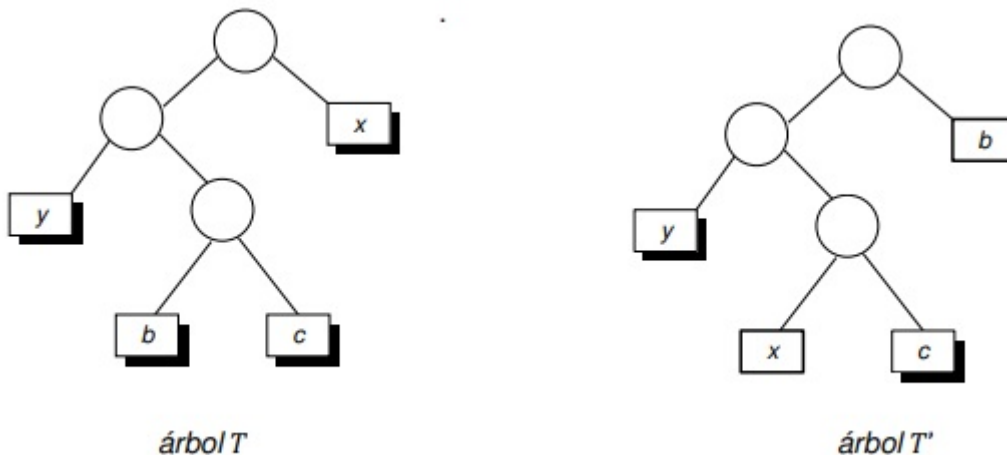
El algoritmo de *Huffman* construye un árbol teniendo en cuenta la frecuencia de cada caracter. En cada paso combina símbolos de menor frecuencia en un nuevo símbolo cuya frecuencia puede expresarse como la suma de estos. Este proceso se hace recursivamente hasta obtener el nodo raíz cuyo símbolo aúna a todos los caracteres. De esta manera se consigue con código sin prefijos.

Demostración: Sea T un árbol binario de codificación prefija óptimo. Sean b y c dos hojas hermanas en T que se encuentran a profundidad máxima. Sean x e y dos hojas de T tales que son los 2 caracteres del alfabeto C con la frecuencia más baja. El caso interesante para la demostración se produce cuando $(b, c) \neq (x, y)$.

T , que es un árbol óptimo, se puede transformar en otro árbol T' , también óptimo, en el que los 2 caracteres, x e y , con la frecuencia más baja serán hojas hermanas que estarán a la máxima profundidad. El árbol que genera el algoritmo cumple exactamente esa condición. Podemos suponer que $f(b) \leq f(c)$ y que $f(x) \leq f(y)$.

También se puede deducir que $f(x) \leq f(b)$ y $f(y) \leq f(c)$.

Construimos un nuevo árbol, T' , en el que se intercambia la posición que ocupan en T las hojas b y c .

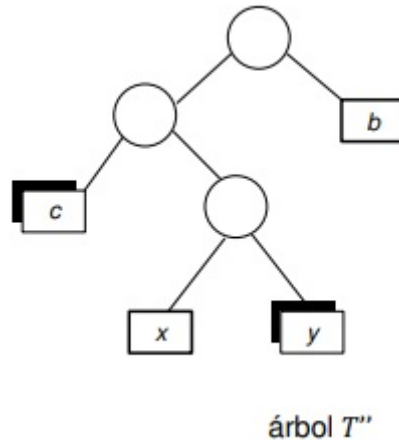


El número de bits para el nuevo árbol de codificación T' lo denotamos por $B(T')$.

Tenemos entonces que: $B(T) - B(T') = \sum_{c \in C} f(c) \cdot d_T(c) - \sum_{c \in C} f(c) \cdot d_{T'}(c) = f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_{T'}(x) - f(b) \cdot d_{T'}(b) = f(x) \cdot d_T(x) + f(b) \cdot d_T(b) - f(x) \cdot d_T(x) - f(b) \cdot d_T(b) = (f(b) - f(x)) \cdot (d_T(b) - d_T(x)) \geq 0$.

De modo que $B(T) \geq B(T')$ pero dado que T es óptimo, mínimo, T' no puede ser menor y $B(T) = B(T')$ por lo que T' también es óptimo.

De forma similar se construye el árbol T'' intercambiando c e y .



Con este intercambio tampoco se incrementa el coste $B(T') - B(T'') \geq 0$. Por tanto, $B(T'') = B(T)$ y como T es óptimo, entonces T'' también lo es.

Algoritmo 2 Construcción de un árbol de Huffman

```

Sea heap un heap de mínimo vacío
para cada caracter  $c$  cuya frecuencia  $f$  hacer
    insertar en heap nodo de etiqueta  $c$  y peso  $f$ 
mientras heap tenga más de un elemento hacer
    árbol_a  $\leftarrow$  extraer(heap)
    árbol_b  $\leftarrow$  extraer(heap)
    etiqueta  $\leftarrow$  Concatenación de las etiquetas de los nodos raíz de árbol_a y árbol_b
    peso  $\leftarrow$  peso(nodo raíz de árbol_a) + peso(nodo raíz de árbol_b)
    árbol_c  $\leftarrow$  árbol de etiqueta etiqueta, peso peso y subárboles izquierdo y derecho árbol_a y árbol_b
    insertar árbol_c en heap
árbol  $\leftarrow$  extraer(heap)
devolver árbol

```

Se detalla en el algoritmo 2 la construcción un árbol de Huffman. Sea n la cantidad de caracteres a utilizar. La construcción de un *heap* de n elementos es de orden $\mathcal{O}(n \log n)$. En cada paso se extraen dos árboles parciales del *heap* y se construye un nuevo árbol a partir de ellos, uniendo sus etiquetas. Ambas operaciones son de orden $\mathcal{O}(\log n)$. Este nuevo árbol es insertado en el *heap*, esta inserción es $\mathcal{O}(\log n)$. Se puede ver que esta operación se llevará a cabo (n) veces. Para ello se considera n caracteres separados por espacios en el orden en el que aparecen en la etiqueta del nodo raíz al finalizar el algoritmo. Es posible representar a una unión como eliminar un espacio en esta cadena. Para unir todos los caracteres es necesario eliminar $n - 1$ espacios presentes. Por ende, la operación de unión se lleva a cabo exactamente $n - 1$ veces, y esto es $\mathcal{O}(n)$. Además, las operaciones de mayor complejidad algorítmica del bucle son la inserción y extracción del *heap*, ambas de orden $\mathcal{O}(\log n)$. Luego el algoritmo es de orden $\mathcal{O}(n \log n)$.

El algoritmo 3 detalla la compresión de una cadena utilizando el árbol creado. Se construye una vector de 256 posiciones que sirva para asociar cada byte posible a su cadena de bits correspondiente al comprimirse. Comprimir cada byte individualmente es obtener un elemento del vector, esto es $\mathcal{O}(1)$. Luego, si la cadena de entrada tiene n bytes, el orden de complejidad temporal del algoritmo de compresión resulta $\mathcal{O}(n)$.

Algoritmo 3 Compresión de un archivo

```

Se construye HT el árbol de Huffman con las frecuencias dadas
Sea v un vector tal que para cada byte b posible v[b] sea el byte b comprimido según HT
bits  $\leftarrow$  arreglo de bits vacío
para cada byte b en el archivo hacer
    Agregar v[b] a bits
devolver bits

```

Un algoritmo de descompresión se detalla en el algoritmo 4. Sea n la cantidad de bits en el archivo que se desea descomprimir. Para cada bit se realiza una operación de orden de complejidad $\mathcal{O}(1)$. Luego la descompresión es de orden $\mathcal{O}(n)$.

Algoritmo 4 Descompresión de un archivo

Sea HT el árbol de Huffman cuyo código fue usado para comprimir el archivo

descomprimido \leftarrow una cadena vacía

árbol_actual \leftarrow HT

mientras cada bit b en el archivo **hacer**

si b es 0 **entonces**

 árbol_actual \leftarrow subárbol izquierdo de árbol_actual

si no

 árbol_actual \leftarrow subárbol derecho de árbol_actual

si árbol_actual es un nodo raíz **entonces**

 agregar la etiqueta de árbol_actual a descomprimido

 árbol_actual \leftarrow HT

devolver descomprimido

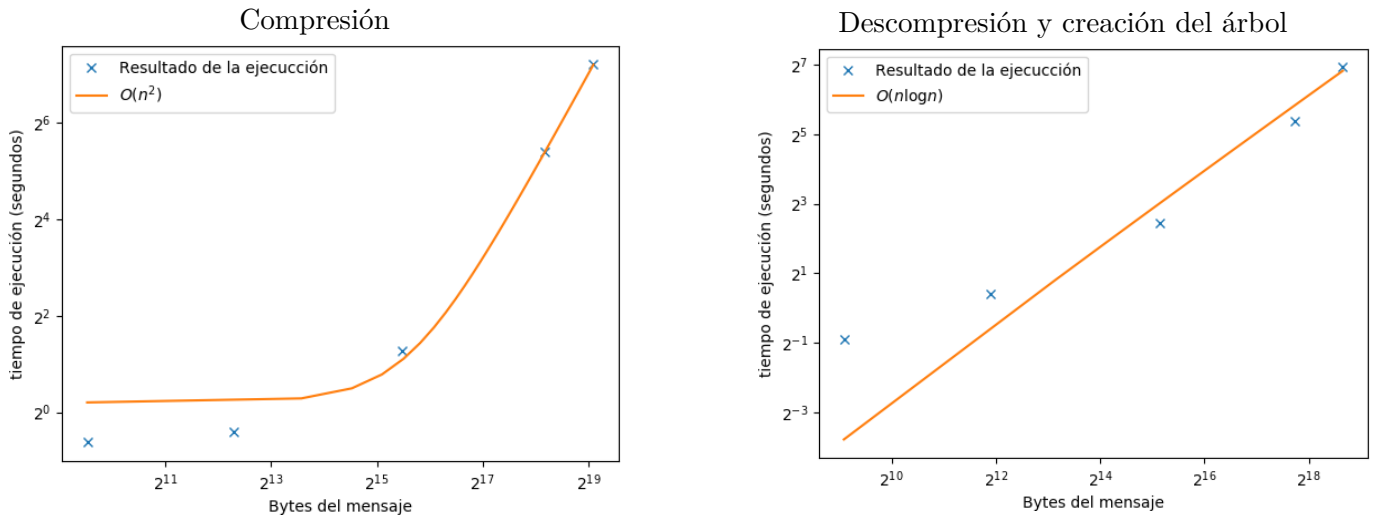


Figura 1: Resultado de los datos relevados.

En la figura 1 se aprecia como si bien la complejidad resultante para el algoritmo de compresión desarrollado es la teórica en el caso de la descompresión esto no es así. Se atribuye la discrepancia a la implementación del arreglo de bits utilizado. En este, las operaciones de inserción y extracción resultan $\mathcal{O}(n)$ en el peor caso.

4. ¿Es el siguiente un algoritmo tractable?

En esta tercera parte se analizó el algoritmo de *Johnson*, “*Finding All The Elementary Circuits Of A Directed Graph*” y se decidió si estamos frente a un algoritmo tractable o no. El orden de complejidad de este algoritmo es $\mathcal{O}((n + e)(c + 1))$ donde n es la cantidad de nodos, e es la cantidad de aristas y c es la cantidad de ciclos que tiene el grafo. Si se analiza el peor caso, donde tenemos un grafo dirigido completo de n nodos, se tienen $e = n(n - 1)$ aristas. La cantidad de todos los subconjuntos formados por los nodos es su cardinal, en este caso 2^n . Como para formar un ciclo se necesitan a lo sumo 2 nodos, para obtener la cantidad de conjuntos que se pueden formar con 2 o mas nodos es la cantidad total restado la cantidad de conjuntos que tiene 1 solo elemento, en este caso $2^n - n$ por esta razón la cantidad de ciclos que puede tener un grafo completo es $c = 2^n - n$ ciclos. En este peor caso la complejidad algorítmica total sería

$$\mathcal{O}((n^2)(2^n - n + 1)) = \mathcal{O}((n^2 \cdot 2^n - n^3 + n^2)) = \mathcal{O}((n^2 \cdot 2^n))$$

Siendo esta complejidad no polinomial no sería tractable. Se puede llegar a decir que este algoritmo pertenece a los problemas Pseudo polinomiales o NP-completos débiles ya que dependiendo de la cantidad de nodos, aristas y forma del grafo podría tener una complejidad polinomial.

Referencias

- [1] Donald B. Johnson. *“Finding All The Elementary Circuits Of A Directed Graph”*.
<https://www.cs.tufts.edu/comp/150GA/homeworks/hw1/Johnson%2075.PDF>
- [2] Jon Kleinberg, Éva Tardos *Algorithm Design* . Addison Wesley,2006.