



Scientia Et Technica  
Universidad Tecnológica de Pereira  
scientia@utp.edu.co  
ISSN (Versión impresa): 0122-1701  
COLOMBIA

2007  
Guillermo Solarte Martínez / Luis Eduardo Muñoz Guerrero  
ALGORITMOS VORACES  
*Scientia Et Technica*, diciembre, año/vol. XIII, número 037  
Universidad Tecnológica de Pereira  
Pereira, Colombia  
pp. 449-454

Red de Revistas Científicas de América Latina y el Caribe, España y Portugal

Universidad Autónoma del Estado de México

<http://redalyc.uaemex.mx>



## ALGORITMOS VORACES

### Algorithms voracious

#### RESUMEN

Los algoritmos voraces son usados esencialmente para resolver problemas de optimización, aunque también pueden aproximarse a una solución a problemas considerados computacionalmente difíciles, Son algoritmos muy fácil de diseñar e implementar y de gran eficiencia.

**PALABRAS CLAVES:** Algoritmos Voraces Prim, Fuman y Kruskal  
Algoritmia, Problemas Np

#### ABSTRACT

*The voracious algorithms are used essentially for solve optimization problems, even also they can approximate to a solution for problem considered difficult computationally, they are algorithms very easy to design and implement and of great efficiency.*

**KEY WORDS:** Prim's voracious algorithms, Fuman and Kruskal Algorithmia, Np Problems.

**GUILLERMO SOLARTE  
MARTINEZ**

Ingeniero de Sistemas.  
Profesor Auxiliar.  
Universidad Tecnológica de Pereira.  
roberto@utp.edu.co

**LUIS EDUARDO MUÑOZ  
GUERRERO**

Ingeniería de Sistemas M.Sc.  
Profesor Auxiliar.  
Universidad Tecnológica de Pereira.  
luismunoz@utp.edu.co

### 1. INTRODUCCIÓN

Este artículo se realiza una exposición de algunos problemas que admiten el uso de esta técnica voraz y los algoritmos que mediante dicha técnica los resuelve.

Para esto utilizamos el lenguaje Phyton para realizar nuestra presentación y para cada uno de ellos se realiza el cálculo de la función de complejidad.

Por ultimo se presenta la teoría de Matroid como soporte teórico para la corrección de algunos algoritmos voraces en la solución de problemas de optimización.

Algunos ejemplos de problemas que se pueden solucionar utilizando un algoritmo voraz son:

- En un grafo ponderando para dar la ruta más corta para ir de un nodo a otro.
- Suponiendo que el sistema monetario de un país esta formado por un numero finito de denominaciones, para cualquier cantidad dad, suministrada en el menor numero posibles de monedas.

En tales contexto, un algoritmo voraz funciona seleccionado la arista o la moneda que parezca más prometedor en un determinado instante, nunca reconsidera su decisión sea cual fuera la situación que pueda surgir más adelante. No hay necesidad de evaluar alternativas, ni de emplear sofisticados procedimientos de seguimientos que permitan deshacer las decisiones

anteriores. Los algoritmos voraces se caracterizan por las siguientes propiedades:

- Existe una función que comprueba si un cierto conjunto de candidatos constituye una solución del problema, ignorando si es o no óptimo por el momento.
- La segunda función, Comprueba si un cierto conjunto de candidatos es factible, es decir si es posible o no completar el conjunto añadiendo otros candidatos para obtener al menos una solución del problema.
- La tercera función, Realiza una selección, que indica en cualquier momento cual es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados.
- Por ultimo existe una función objetivo queda el valor de la solución hallada, por ejemplo la longitud de la ruta que se ha construido o el número de monedas utilizadas para cambiar una cantidad, a diferencia de las funciones mencionadas anteriormente, la función objetivo no aparece explícitamente en el algoritmo voraz, algunas veces la función de selección suele estar relacionada con la función objetivo, por ejemplo si se intenta maximizar es posible que se seleccione el candidato restante que posea mayor valor individual, como el problema de cambio de moneda; si por el contrario se intenta minimizar el coste, quizá se seleccione de los candidatos disponibles el menor valor. Sin embargo, en algunas ocasiones puede haber varias

funciones de selección disponibles así que hay que escoger la adecuada para lograr que el algoritmo funcione correctamente.

Para resolver un problema, se busca un conjunto de candidatos que constituya una solución, y que optimicé (minimice o maximice, según el caso), el valor de la función objetivo. Los algoritmos voraces avanzan paso a paso, la solución es buscar entre los subconjuntos del conjunto inicial o del conjunto candidato.

Inicialmente el conjunto seleccionado es vacío. Entonces en cada paso se considera adicionar a este conjunto el mejor candidato sin considerar los restantes. Una vez seleccionado el nuevo elemento, se verifica si el conjunto con el conformado es el prometedor, es decir que puede conducir a una solución. Si el conjunto es factible el elemento se incorpora al conjunto solución y permanece allí hasta al final; si el conjunto no es prometedor, el elemento se rechaza y no vuelve a ser considerado. Cada vez que se amplía el conjunto de candidatos seleccionados, se verifica si éste constituye una solución para el problema; si lo es termina el algoritmo, de lo contrario se considera un nuevo elemento este proceso se realiza varias veces hasta que se hayan considerado todos los candidatos.

Estructura General de Un algoritmo Voraz en python

```
def voraz (C):
    S=0 // en s se almacena los elementos de la solución
    while ((C != 0) and not( solucion (S) ):
        x= seleccionar (C)
        C = C - {X}
        if factible (S U {X}):
            S = S U {X}
    return S
```

#### Funciones genéricas:

- **Solucion.** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- **Seleccionar.** Devuelve el número más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
- **Factible.** Indica si a partir del conjunto S y añadiendo X, es posible construir una solución (posiblemente añadiendo otros elementos).

Algunos problemas que se ajustan al esquema voraz

#### Problema del Árbol de recubrimiento mínimo

Dado un grafo conexo no dirigido y ponderado, el problema consiste en encontrar un árbol de recubrimiento de G cuyo peso sea mínimo. Formalmente,

- **INSTANCIA:**  $G = (V, A)$ , donde G es un grafo ponderado no dirigido.
- **SOLUCIÓN:** Un subconjunto SD de A con exactamente  $|V|-1$  aristas, tal que no contiene ciclos.
- **MEDIDAS:**  $m(S) = \sum P(e)$ , siendo  $p(e)$  el peso correspondiente a la arista e.
- **OBJETIVO:** Minimizar  $m(S)$ .

#### Lema 1 de corrección

Sean  $G = (V, A)$  un grafo ponderado, conexo y no dirigido,  $B \subset V$  un subconjunto estricto de G,  $S \subseteq A$  un conjunto de arista prometedor tal que toda arista de S tiene sus vértices en B, y e la arista de menor peso que tenga un vértice en B y otro en  $V - B$ . Entonces  $S \cup \{e\}$  es prometedor.

**Demostración.** Sea G, B Y e como en la hipótesis; se U un árbol de recubrimiento mínimo tal que  $S \subseteq U$ . Este U tiene que existir ya que S es prometedor por hipótesis; si  $e \in U$ , no hay nada que probar puesto que se podría seguir adicionando las demás arista de U para finalmente tener  $S = U$ .

En caso contrario,  $U \cup \{e\}$  contiene exactamente un ciclo, además debe existir otras arista  $u$  que tenga un vértices en B y otro en  $V - B$  (de otra manera el ciclo no se cerraría), ahora si se elimina  $u$  de U, el ciclo desaparece y se obtiene un nuevo árbol de U' que recubre a G, pero el peso de  $u$  no es mayor que el de e por definición. así el peso de U' no supera el de U por lo tanto U' es también un árbol de recubrimiento mínimo de g de esta forma  $S \cup \{e\} \subseteq U'$  y  $S \cup \{e\}$  es prometedor.

#### Algoritmo Prim

**El algoritmo Prim [1].** Para resolver este problema se empieza a construir el árbol de recubrimiento a partir de un nodo arbitrario de V. Luego, en cada interacción se inserta al árbol la arista de menor peso tal que ésta conecta el árbol ya construido con un vértice fuera de él; su trabajo termina cuando el árbol contiene todos los vértices V. En el siguiente algoritmo B almacena los vértices de árbol solución y S el conjunto de arista que lo conforman.

```
def prim (G = (V,A),p): conjunto de arista
    B: conjunto de vértices
    S: conjunto de arista
    S= 0
    B= // Un miembro arbitrario de V
```

```
While B != V :
    if  $e = \{u,v\} \in A$ , tal que  $u \in B$  y  $v \in V - B$ 
    Con  $p(e)$  de peso mínimo
    S= S  $\cup$  {E}
    B= B  $\cup$  {v}
return S
```

**Demostración**

Esta demostración se hará por Inducción Matemática sobre los números de arista del conjunto S. Se demostrará que si el conjunto S es prometedor en alguna instancia del algoritmo, al agregarle una arista adicional éste seguirá siéndolo. Así, después de ser insertadas n-1 arista a S, éste será una solución al problema que además es óptima, ya que S es prometedor. Para  $S = \emptyset$ ,  $\emptyset$  es prometedor ya que a este se le puede adicionar las aristas de cualquier de los árboles que recubren a G, en particular las de algún árbol de recubrimiento mínimo.

Supóngase que S es prometedor y que  $B \subset V$  es el conjunto de vértices de las arista de S; ahora como e es la arista de menor en A que tiene un único vértice en B por definición, se satisfacen las hipótesis del lema 1 se tiene que  $S \cup \{e\}$  es un conjunto prometedor; esto completa la demostración de que en cualquier instancia del algoritmo S es prometedor, así cuando termina S constituye una solución óptima al problema.

Para facilitar la implementación del Algoritmo de Prim, se dispone que los nodos del grafo están

<sup>[1]</sup> Este nombre se debe al matemático estadounidense Robert C. Prim (1921) quien publicó este en 1957.

para representar el grafo de la siguiente manera  $G.mat[i,j]$  contiene el peso de la arista  $\{i,j\}$  o  $\infty$ [2] si esta arista no existe, de esta manera, la función prim tiene como única entrada el grafo G. Este algoritmo requiere además el uso de dos vectores masprox y dminima tales que  $masprox.vec[i]$  representa el nodo más cercano al nodo i y  $dminima.vec[i]$  el peso de la arista  $(i,masprox.vec[i])$  para todo nodo  $i \in V - B$ , siendo V el conjunto de todos los nodos del grafo y B el conjunto de nodos que hacen parte del árbol de recubrimiento que se está construyendo.

```
from record import record
clases arista(record)
v1,v2 cardinal
Altura = arista
N cardinal
Vec= [1..max]
grafo = arista
```

```
N:cardinal
Mat = Matriz[1..max][1..max] of cardinal
Conjuntoaristas = conjunto de arista
```

**Función prim (G):**

```
i, minimo: cardinal
Aux: arista
Masprox , dminima: alturas
S: conjuntosaristas
1 S = 0
2 for i in range(2, G.n):
3   masprox.vec[i] = 1
4   dminima.vec[i] = G.mat[1,i]
```

```
5 for i in range(1, G.n):
6   minimo =  $\infty$ 
7   for j in range(2, G.n):
8     if  $(0 \leq dminima.vec[j] \text{ and } 0 \leq minimo)$ :
9       minimo = dminima.vec[j]
10      k = j
11 S =  $S \cup \{masprox.vec[k],k\}$ 
12 dminima.vec[k] = -1

13 for j in range(2, G.n):
14   if  $(G.mat[j,k] < dminima.vec[j])$  :
15     dminima.vec[k] = G.mat[j,k]
16     masprox.vec[j] = k
17 return S
```

Sea n el número de nodos del grafo, la siguiente tabla describe el calculo de la función de complejidad de la función prim:

<sup>[2]</sup> En la práctica,  $\infty$  es un valor mayor que el peso de cualquier arista del grafo.

3- 4	2
2- 4	$3n-2$
1-4	$3n-1$
8- 10	4
7 -10	$5n-4$
6-10	$5n-3$
6-12	$5n$
14-16	$4n-3$
6 -16	$9n-3$
5-16	$9n^2 -3n + 1$
1-17	$9n^2 + 6n - 2$
f(n) =	$9n^2 + 6n - 2$

Tabla 1. Ejemplo de complejidad.

Lo anterior indica que la función de complejidad de función prim está en  $O(n^2)$ .

**Algoritmo de Kruskal**

El algoritmo de Kruskal[3] soluciona el problema del árbol de recubrimiento mínimo ordenado, primero, él peso de la arista del grafo de forma ascendente; el algoritmo crea un bosque (conjunto de árboles) donde cada vértice o nodo del grafo es un árbol separado. El árbol de recubrimiento empieza con un conjunto solución vacío en el cual se añade en cada paso la arista de menor peso de tal forma que no se produzca ciclos, para añadir una arista, se evalúa si ésta une dos árboles distintos, de lo contrario no se considera como parte de la solución. Nótese que en cada fase el número de árboles del bosque se reduce en uno. Al final quedará conformado un solo árbol, éste será el árbol de recubrimiento mínimo para todos los nodos del grafo.

Para la implementación del algoritmo se utilizarán los siguientes tipos de datos

```
from record import record
clases arista(record)
vi,v2: cardinal
peso:cardinal

arista = record
na:cardinal
Vec = vector[1..max] of arista
Caris = conjunto de arista
Vcar = registro
n: cardinal
Vec = vector[1..max] of cardinal
```

<sup>[3]</sup> Es un algoritmo de la teoría de grafos para encontrar un árbol expandido mínimo en un grafo conexo y ponderado...

de este iterada á una estructura de conjuntos disjuntos de tal forma que cada nodo esta en un único conjunto. Sabiendo que inicialmente el bosque generado por la arista de la solución contiene  $n$  árboles de un sólo nodo y que cada vez que se inserta una arista a la solución se unen los árboles que contienen sus vértices, se define tales árboles como sigue, los nodos estarán representados por un único índice  $i$  en el vector conjunto, de tal forma que conjunto.vec[i] contiene el índice correspondiente al padre de nodo  $i$ , en caso de que conjunto.vec[i]  $\neq i$ , ó el nodo correspondiente al índice  $i$  sea la raíz del árbol al cual pertenece, en ese caso el conjunto.vec[i] =  $i$ .

Así mismo, se define el vector altura, de manera que altura.vec[i] almacenará la altura del árbol al cual pertenece el nodo  $i$ , para todo  $i$  con conjunto[i] =  $i$ .

Con el objetivo de efectuar la unión de conjuntos se define el procedimiento fusionar.

```
def fusionar(va ,vc ,r1,r2 ):
    i : cardinal
    1 if (va.vec[r1]= va[r2])
    2 inc(va.vec[r1])
    3 vc.vec[r2] = r1
    else:
    4 if (va.vec[r1] > va.vec[r2])
    5 vc.vec[r2] = vevc[r1]
    else
    6 vc.vec[r1] = vc.vec[r2]
```

El procedimiento fusionar realiza tres operaciones en el peor caso, es decir que su tiempo de complejidad es constante.

```
def buscar (vconj k ):
    i : cardinal
    i= k
    while ( i != vconj.vec[i]):
        i = vconj.vec[i]
    return i
```

La función buscar en la segunda línea ejecuta tanta veces el ciclo como la altura árbol al cual pertenece el nodo  $k$ , como se prueba más adelante es como máximo  $\lceil \log ( |vconj| ) \rceil$ , donde  $|vconj|$  representa el número de elementos de  $vconj$ ; así en el ciclo se efectúan un total de  $2\log(|vconj|) + 1$  y en la función un total de  $2\log(|vconj| + 3)$  operaciones elementales.

```
def kruskal (n A):
    i,u,v,contador cardinal
    Conjunto, altura alturas
    Solucion : Caris
    1 solucion = " "
    2 contador = 0
    3 for i in range(1, n):
    4 Conjunto.vec[i] = i
    5 altura.vec[i] = 0

    6 ordenar_por_monticulo(A,a,na )
    7 i= 1
    8 while (contador < n):
    9 u= buscar (A.vec[i].v1,conjunto)
    10 v= bucar_(A.vec[i].v2,conjunto)
    11 if u != v :
    12 solucion = solucion  $\cup$  {A[i]}
    13 contador++
    14 fusionar(altura, conjunto, u, v)
    15 i++
    16 return solucion
```

La función de complejidad de la función Kruskal, en término del número de nodos y arista del grafo, esta dada por:

Líneas	Numero de Operaciones
4-5	2
3-5	$3n+1$
1-5	$3n+3$
1-7	$3n + n\log(na)+4$
9-15	$2\log(n) + 8$
8-15	$2n\log(n)+2\log(n) + 10n-8$
1-16	$2n\log(n)+n\log(na)-2\log(n)+13n-3$
f(n,na)=	$2n\log(n)+n\log(na)-2\log(n)+13n-3$

Tabla 2 función de complejidad

es decir, la función de complejidad de Kruskal está en el  $O(n\log(na))$ , pero como se sabe:

$$n-1 \leq na \leq n(n-1)/2$$

$$i \leq na / (n-1) \leq n/2$$

$$\log 1 \leq \log ( n/(n-1) ) \leq \log(n/2)$$

$$0 \leq \log (na) - \log(n-1) \leq \log(n) - \log 2$$

$$\log(n-1) \leq \log(na) \leq \log(n) + \log(n-1) - \log 2$$

$$\log(n-1) \leq \log(na) \leq 2\log(n) - \log 2$$

Esto es,  $\log(na) \in O(\log(n))$ , por consiguiente  $n\log(na) \in O(n\log(n))$ .

Como se probó anteriormente, los algoritmos de Prim y Kruskal tienen funciones de complejidad que están en el

<sup>[4]</sup>En honor al Ingeniero Eléctrico estadounidense David A H. (1925-1999) quien publicó este algoritmo en 1952

orden exacto  $n^2$  y  $n \log(n)$  respectivamente; luego como  $n-1 \leq na \leq n(n-1)/2$  la función de complejidad de Kruskal acota superiormente a la prim, cuando  $na$  es próximo a  $(n-1)n/2$  y viceversa cuando  $na$  es cercano a  $n-1$ .

**CODIGO DE HUFFMAN**

La codificación de Huffman es una técnica ampliamente usada y muy efectiva para la compresión de datos, esta técnica reduce en gran porcentaje el espacio en memoria de un archivo, tal reducción depende de las características del mismo.

El problema de los códigos de Huffman[4] consiste en determinar un código binario[5] para representar cada uno de los caracteres, de tal manera que el número de bits requerido para representar el texto sea mínimo; éste problema se define formalmente de la siguiente manera:

**INSTANCIA:** los caracteres  $c_1, c_2, c_3, \dots, c_n$  y sus frecuencias  $f_1, f_2, \dots$

**SOLUCION:** Un conjunto  $S$  de códigos binarios  $cod(c_1), cod(c_2), \dots, cod(c_n)$

**MEDIDAS:**  $m(S) = \sum_{i=1}^n f_i |cod(c_i)|$  donde  $|cod(c_i)|$

es la longitud del código binario  $cod(c_i)$ .

**OBJETIVO:** Minimizar  $m(S)$

La solución que se encuentra esta representada por un árbol binario<sup>5</sup> de la siguiente manera:

- Las hojas del árbol son los caracteres
- Al recorrer el camino de la raíz a una hoja determinada se obtiene el código de dicha hoja, con la interpretación 0 si el siguiente nodo del camino es hijo izquierdo y 1 si es hijo derecho.
- $|cod(c)|$  corresponde a la profundidad del carácter  $c$ .

Tipo  $pcar = \uparrow$  car este puntero almacena la dirección de memoria de una variable car  
 Car = registro  
 FREC: cardinal  
 Hd,hi:pcar  
 Fin\_registro  
 Cars= registro  
 N:cardinal  
 Vec:vector[1..max]de pcar  
 Fin de registro  
 Vecval=registro  
 N:cardinal  
 Vec:vector[1..max] de cardinal  
 Fin\_registro

La siguiente función extrae el menor elemento de montículo  $C$ , conservando esta estructura en  $C$

```
def extraermin ( C ):
    i,menor: cardinal
    1 extraermin = C.vec[1]
    2 intercambiar (C.vec1,C,n)
    3 dec(C.n)
    4 hundir(C.vec,C,n,1)
```

la función  $extraermin$  tiene una complejidad de  $O(\log(n))$ .

El procedimiento  $insertar$  introduce el elemento  $x$  al final de  $C$ , y posteriormente lo hace flotar para mantener la propiedades de montículo,

```
def insertar( C, x):
    1 inic(C.n)
    2 C.vec[C.n] = x
    3 Flotar (C.vec,c,n)
```

El procedimiento tiene una complejidad de  $\log(n)+2$ .

La función Huffman inicialmente crea un montículo con el vector de caracteres  $C$ , paso seguido extrae los dos elementos de menor peso del montículo, luego crea un nuevo caracter que tenga a éstos como hijos y cuyo peso sea la suma de los pesos de sus hijos.

Por último, el nuevo carácter es insertado en el montículo, este proceso es realizado  $n-1$  veces, hasta haber conformado el primer caracter de  $C$  un árbol binario que representa la codificación de los caracteres iniciales.

```
def Fuman ( c ):
    i: cardinal; aux: pcar
    1 Craemonticulo_de_minimos( C.vec)
    2 For i in range( Cn-1 ):
    3     if ( nuevo (aux))
        hi = extraermin(C)
        hd = extraermin(C)
        frec= hi↑.frec+hd↑.frec
    6     return C.vec[1]
```

Su análisis es presentado a continuación:

Líneas	Números de operaciones
5-8	$3 \log(n) + 4$
4-8	$3n \log(n) - 3 \log(n) - 3$
1-8	$3n \log(n) - 3 \log(n) + 3n + 1$
F(n)	$3n \log(n) - 3 \log(n) + 3n + 1$

Tabla 3 función de complejidad

Por lo tanto  $f(n) = O(n \log(n))$  es la función de complejidad de este algoritmo

**CONCLUSIONES**

Para el problema del árbol de recubrimiento mínimo, la relación entre la complejidad del algoritmo de Prim y la del algoritmo de Kruskal dependen fundamentalmente de la cantidad de aristas del grafo; más concretamente, el algoritmo Prim es más eficiente cuando el número de grafo tiende a ser complejo y el algoritmo de Kruskal

cuando el número de arista del grafo se aproxima  $n-1$ , siendo  $n$  el número de nodos del grafo.

Aquellos algoritmos voraces dependen de la instancia para encontrar una solución óptima a cierto problema, es decir que no siempre encuentran una solución óptima, en los casos en que resuelven el problema correctamente, son algoritmos muy eficientes.

## **BIBLIOGRAFIA**

- [1]. CORMEN, Thomas ,LEISERON, Charles y Riverts, Ronald, Introducion to algoritmo. Nueva Cork Mc GRAWhill.1990
  
- [2]. MARTIN , James ODELL James J Analisis y diseño orientado a objetos, Mexico Prentice Hall Hispanoamericana 1994.
  
- [3]. G.BRASSARD,P Bratey Fundamentos de Algoritmia Madrid Prentice 1999.  
Algoritmos voraces disponibles en Internet  
<http://www.isi.upv.es/evidadl/students/ad3/tema4>  
<http://www.isi.upc.edu/eia/transpasjavier/voracesjavier.ptt>