ALGORITMOS VORACES

WILFREDO YATE TORRES
YADDY MELISSA SÁNCHEZ OFO

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA EDUCACIÓN
DEPARTAMENTO DE MATEMÁTICAS
POPAYÁN, CAUCA
2006

ALGORITMOS VORACES

WILFREDO YATE TORRES
YADDY MELISSA SÁNCHEZ OFO

Presentado como requisito parcial para
optar a los títulos de
MATEMÁTICO Y LICENCIADA EN EDUCACIÓN CON
ESPECIALIDAD EN MATEMÁTICAS

Director:
Mg. MAURICIO MACA CHAGÜENDO

UNIVERSIDAD DEL CAUCA
FACULTAD DE CIENCIAS NATURALES, EXACTAS Y DE LA
EDUCACIÓN
DEPARTAMENTO DE MATEMÁTICAS
POPAYÁN, CAUCA
2006

Nota de aceptación

_____

_____

_____

_____

Director: Magister Mauricio Maca Chagüendo

_____

Jurado: Especialista Diego Ramiro Correa

_____

Jurado: Doctor Fredy Amaya

Fecha de sustentación: Popayán, junio 12 de 2006.

A nuestros padres y hermanos

AGRADECIMIENTOS

Deseamos manifestar nuestros más sinceros agradecimientos a todas aquellas personas que de alguna manera han hecho posible la consecución de este logro, especialmente a:

María Delia Correa, Rocío de las Mercedes Torres, Francisco Fernando Bohorquez y Germán Rolando Burbano por su significativo respaldo durante nuestros estudios.

Los profesores que contribuyeron con nuestra formación, en particular a Mauricio Maca, Diego Correa y Fredy Amaya por su valiosa colaboración, atención, paciencia y sincera amistad.

Nuestros familiares por su permanente motivación e incondicional apoyo.

Nuestros amigos por su preciada compañía y múltiples manifestaciones de afecto.

CONTENIDO

NOTACIÓN

$A \subseteq B$	A es subconjunto de B
$A \times B$	producto cartesiano A por B
$\mathcal{P}(X)$	partes de X
$x \in A$	x es elemento de A
$x \notin A$	x no es elemento de A
$\emptyset$	conjunto vacío
$\mathbb{R}$	conjunto de los números reales
$\mathbb{R}^+$	conjunto de los números reales positivos
$\mathbb{R}^n$	$\underbrace{\mathbb{R} \times \mathbb{R} \times \ldots \times \mathbb{R}}_{n-veces}$
$\mathbb{N}$	conjunto de los números naturales
$\lfloor x \rfloor$	parte entera de x
$\log(x)$	logaritmo de x en base 2
div	división entera
mod	módulo
$dec(x, k)$	decrementa x en k unidades
$dec(x)$	decrementa x en 1
$inc(x)$	incrementa x en 1
$nuevo(x)$	crea una variable, y asigna a x la dirección en memoria de la variable creada

INTRODUCCIÓN

Los algoritmos voraces son usados generalmente para resolver problemas de optimización, aunque también pueden aproximar una solución a problemas considerados computacionalmente difíciles, son fáciles de diseñar e implementar, además de ser algoritmos de gran eficiencia; por otro lado, la técnica voraz utilizada por ellos constituye una herramienta útil cuando se enfrenta un problema.

Este documento presenta algunos de los problemas más comunes que admiten el uso de la técnica voraz y los algoritmos que mediante dicha técnica los resuelven, el seudocódigo utilizado para la presentación de estos algoritmos corresponde al asociado al lenguaje de progamación Pascal y para cada uno de ellos se realiza el cálculo de la función de complejidad. El presente trabajo incluye además de la presentación teórica, una aplicación computacional que facilita la comprensión en detalle del funcionamiento de esta técnica en la solución de cada uno de los problemas presentados. Este software fué implementado en el lenguaje de programación Delphi y presenta de forma dinámica e interactiva el proceso, paso a paso, que efectúa cada uno de los algoritmos voraces presentados para hallar la solución del problema que resuelve con instancias determinadas por el usuario. Se anexa a este documento un cd-room que contiene el software mencionado anteriormente.

Por último se exhibe además la *teoría de matroid* como soporte teórico para la corrección de algunos algoritmos voraces en la solución de problemas de optimización.

Capítulo 1

PRELIMINARES

Teniendo en cuenta que el propósito principal de este trabajo es realizar un estudio detallado de la técnica de programación **voraz**, en esta sección se exponen las bases fundamentales para el desarrollo de dicho estudio.

1.1. ALGORITMO

La palabra **algoritmo** se debe al matemático persa del siglo IX Al-Khowârizmî. Un algoritmo determinístico se define como una secuencia finita de instrucciones que conduce a la solución de algún problema. La definición de un algoritmo debe describir un conjunto entrada, un proceso y un conjunto salida; además, este debe satisfacer las siguientes características fundamentales:

- Debe ser preciso e indicar el orden de realización de cada instrucción.
- Debe ser definido, es decir, en cada uno de los casos del problema que dice resolver, debe funcionar correctamente.
- Debe ser finito, es decir, para cualquier conjunto de entrada el algoritmo efectúa un número finito de instrucciones.

Así mismo, en el ámbito computacional un algoritmo se puede ver como una herramienta para resolver un problema computacional específico.

1.2. PROBLEMA

En los cursos básicos de programación de computadores se adquiere la noción de que estas máquinas se usan para ejecutar algoritmos en la solución de algunos problemas. Actualmente, los problemas que se quieren resolver por computador pueden tener características variables. En general, se puede expresar el **problema** como una relación $\boldsymbol{P} \subseteq \boldsymbol{I} \times \boldsymbol{S}$, donde $\boldsymbol{I}$ y $\boldsymbol{S}$ son los conjuntos de casos particulares y de soluciones, respectivamente. Como una alternativa para ver el problema, se puede considerar también una proposición $\boldsymbol{p}(x, y)$ la cual es verdadera si y sólo si $(x, y) \in \boldsymbol{P}$. Si se quiere analizar las propiedades de los cómputos realizados, es necesario considerar las características de los conjuntos $\boldsymbol{I}$, $\boldsymbol{S}$ y de la relación $\boldsymbol{P}$ (o de la proposición) específicamente.
En algunos casos se desea determinar si una instancia x satisface una condición dada. Esto pasa, por ejemplo cuando se quiere verificar si un programa es sintácticamente correcto o un cierto número es primo, o cuando se usa la prueba de un resultado para decidir si una fórmula lógica es un teorema en una teoría dada. En todos esos casos, la relación $\boldsymbol{P}$ se reduce a una función f: $\boldsymbol{I} \rightarrow \boldsymbol{S}$, donde $\boldsymbol{S}$ es el conjunto binario $\boldsymbol{S}$ = {si, no} (o $\boldsymbol{S}$ = {0,1}), y se denota el problema como un "problema de decisión"; también se pueden considerar problemas de "búsqueda", donde para cada instancia $x \in \boldsymbol{I}$, es retornada una solución $y \in \boldsymbol{S}$ tal que $(x, y) \in \boldsymbol{P}$. Esto incluye problemas como, por ejemplo, encontrar en un grafo un camino entre dos nodos dados o determinar la factorización de un entero. En otros casos cuando dada una instancia $x \in \boldsymbol{I}$, se quiere encontrar la "mejor" solución y' entre todas las soluciones $y \in \boldsymbol{S}$ tal

que $\mathbf{p}(x,\ y)$ se verifica. Por ejemplo, cuando dados un punto p y un conjunto de puntos Q en el plano se quiere determinar el punto $q \in Q$ más cercano a p, o cuando dado un grafo con peso se desea hallar un ciclo Hamiltoniano de costo mínimo. Problemas de este tipo son llamados problemas de "optimización" y se presentan frecuentemente en muchas actividades humanas desde el inicio de la historia de las matemáticas.

1.3. PROBLEMA DE OPTIMIZACION

Un problema de optimización P se caracteriza por la siguiente cuádrupla de objetos (I_p, Sol_p, m_p, $Objetivo_p$), donde:

1. I_p es un conjunto de instancias de P.
2. Sol_p es una función que asocia a cualquier instancia de entrada $x \in I_p$ el conjunto de soluciones factibles de x.
3. m_p es la función medida, definida por pares (x, y) tal que $x \in I_p$ e $y \in Sol_p(x)$. Para cualquier par (x, y), $m_p(x, y)$ proporciona un valor positivo el cual corresponde a la solución factible y.
4. $Objetivo_p$ especifica si P es un problema de maximización o minimización.

Dada una instancia de entrada x, se denota por $Sol_p^*(x)$ al conjunto de soluciones de x cuyo valor es óptimo. Formalmente para cualquier $y^* \in Sol_p^*(x)$:

$$m_p(x, y^*) = \text{mín}\{\ v \mid v = m_p(x, z) \text{ y } z \in Sol_p(x)\}$$

en caso de que $Objetivo_p$ sea minimizar y

$$m_p(x, y^*) = \text{máx}\{\ v \mid v = m_p(x, z) \text{ y } z \in Sol_p(x)\}$$

en caso de que $Objetivo_p$ sea maximizar.
El valor de cualquier solución óptima y^* de x será denotada como $m_p^*(x)$.
En adelante, para referirnos al problema, se omitirá el subíndice p.

1.4. ESTRUCTURAS DE DATOS

La información que proviene del mundo real no es directamente entendible por un computador y por tanto debe ser transformada al lenguaje de la máquina, es decir, a un conjunto de ceros y unos. Se debe elegir entonces, una representación adecuada que permita realizar tal abstracción entre mundo real y el computador. Las estructuras de datos tienen como objetivo fundamental la optimización de la representación de los datos teniendo en cuenta dos factores:

- Almacenamiento eficiente en memoria.
- Acceso rápido a la información almacenada.

El manejo de las estructuras de datos es determinante para la consecución de algoritmos eficientes. A continuación se presentan las estructuras de datos utilizadas con mayor frecuencia, así como algunas estructuras más complejas.

1.4.1. Arreglo

Un arreglo es una estructura de datos formada por una cantidad fija de datos de un mismo tipo, cada uno de los cuales tiene asociado uno o más índices que determinan de forma unívoca la posición del dato en el arreglo. Un arreglo permite abstraer o modelar implícitamente problemas, así como distribuir y representar datos. Se puede ver un arreglo como una estructura de celdas donde se puede almacenar información.
Los arreglos más utilizados son los **vectores** (arreglos unidimensionales)

y las **matrices** (arreglos bidimensionales). Ejemplo:
arreglo[1..50] de enteros, es un arreglo unidimensional y
arreglo[1..30, 1...40] de enteros, es un arreglo bidimensional.

1.4.2. Registro

Un **registro** es una estructura de datos conformada por un número fijo de items llamados *campos* que contienen información relativa a un mismo ente; cada uno de los campos puede ser de diferente tipo y para definir un registro es necesario especificar el nombre y el tipo de cada campo.

1.4.3. Grafo simple no dirigido

Un **grafo simple no dirigido**[1] G es un par de conjuntos finitos (V, A), tal que A es una relación binaria simétrica en V llamada conjunto de *aristas* de G y V es un conjunto de *vértices* (*nodos*). La arista $\{u, v\}$ identifica tanto la dupla (u, v) como la dupla (v, u).
Dados $u, v \in V$, si $e = \{u, v\} \in A$ se dice que u y v son adyacentes y que e es incidente en u y en v.
Gráficamente un grafo se representa por medio de un diagrama, con los nodos mostrados como círculos y la aristas como segmentos que conectan los nodos que inciden.

El **grado** de un nodo es la cantidad de aristas incidentes en él. El *grado del grafo* G es el máximo grado de todos los nodos de G; si todos los nodos de G tienen igual grado se dice que G es *regular*.

Un grafo $G = (V, A)$ es **ponderado** si existe una función de peso

[1] En adelante se referirá a él, simplemente como **grafo**.

$f : A \to \mathbb{R}^+$.

Se dice que un grafo $G = (V, A)$ es **completo** si todos los nodos son adyacentes entre si.

Dado un grafo $G = (V, A)$, y $v_0, v_k \in V$, un **camino** de v_0 a v_k es una sucesión de vértices $v_0, v_1, \ldots, v_k$ tal que, $\{v_i, v_{i+1}\} \in A$, para todo i con $1 \leq i \leq k-1$. Tal camino es **simple**, si $v_i \neq v_j$ para $0 \leq i, j \leq k$ con $i \neq j$, es decir, si no se repite ningún vértice en la sucesión.

Un **ciclo** es un camino que satisface $v_0 = v_k$, es decir que empieza y termina en el mismo nodo. Si un grafo no contiene ciclos, se dice que es un grafo **acíclico**.

Un **ciclo simple** es un ciclo en donde $v_i \neq v_j$ para $2 \leq i, j \leq k-1$ con $i \neq j$. en otros términos, los únicos nodos iguales en el ciclo simple son el primero y el último.

Se dice que un grafo es **conexo** si para cualquier par de nodos distintos $u, v \in V$, existe al menos un camino de u a v.

1.4.4. Árbol

Un **árbol** es un grafo no dirigido, acíclico y conexo. Bajo estas condiciones, un árbol $T = (V, A)$ posee las siguientes propiedades:

- $|A| = |V| - 1$
- Si se agrega una única arista e a A, tal que $e \notin A$, entonces, T contiene un único ciclo.

- Si se extrae una única arista e de A, entonces, T deja de ser conexo.
- Un único nodo se considera un árbol.

1.4.5. Árbol con raíz

Si en un árbol se ha establecido una relación de jerarquía (padre-hijo) para cada par de nodos adyacentes, de tal manera que cada nodo tenga un único padre (si lo tiene) y varios hijos en caso de que tenga, además, que exista un único nodo sin padre, llamado *nodo raíz* entonces, se dirá que éste es un **árbol con raíz**. Los nodos sin hijos serán llamados *hojas* del árbol.

La **altura** de un nodo se define como la cantidad de aristas del camino más largo desde este nodo hasta alguna hoja, dicho camino debe contener alguno de los hijos del nodo en caso de que tenga. Así, si este nodo es una hoja, su altura es cero.

La **profundidad** de un nodo es la cantidad de aristas del camino desde la raíz hasta dicho nodo.

1.4.6. Árbol binario

Un árbol **binario** es un árbol con raíz cuyos nodos tienen a lo más dos hijos. Si un nodo tiene dos hijos al primero se le llama hijo izquierdo, y al segundo hijo derecho; si tiene un solo hijo, este puede estar situado como hijo izquierdo o como hijo derecho.

1.4.7. Árbol binario completo

Se dice que un árbol binario es **completo** si cada uno de sus nodos tiene dos o ningún hijo, y todas sus hojas tienen la misma profundidad.

1.4.8. Árbol binario semicompleto

Un árbol binario **semicompleto** de n nodos se construye a partir del árbol completo de $n+q$ nodos, quitando q hojas de derecha a izquierda. De esta definición se desprende el siguiente resultado:

Proposición 1.4.1. *Un árbol binario semicompleto de n nodos tiene altura $\lfloor \log(n) \rfloor$.*

Demostración. Sea k la altura del árbol, el árbol tiene un único nodo con altura k(el nodo raíz), tiene 2 nodos con altura $k-1$ (los dos hijos de la raíz), y así sucesivamente tiene 2^{k-1} nodos de altura 1 y al menos uno pero no más de 2^k nodos con altura 0, de esta manera,

$$\sum_{i=0}^{k-1} 2^i + 1 \leq n \leq \sum_{i=0}^{k} 2^i$$

$$(2^k - 1) + 1 \leq n \leq 2^{k+1} - 1$$

$$2^k \leq n < 2^{k+1}$$

$$k \leq \log n < k + 1$$

$$k = \lfloor \log n \rfloor$$

como se quería probar. □

1.4.9. Montículo

Un montículo[2] de máximos (mínimos) es un árbol binario semicompleto en el que la información almacenada en cada nodo es mayor (menor) o igual que la almacenada por sus hijos, esta condición es llamada propiedad de montículo.
Un árbol binario semicompleto de n nodos se puede representar mediante un arreglo M con n componentes de la siguiente manera:

[2]se conoce por su nombre en inglés "Heap".

- $M[1]$ es la raíz del árbol.
- $M[2i]$ es el hijo izquierdo de $M[i]$ para $1 \leq i \leq (n\ div\ 2)$ e $i \in \mathbb{N}$.
- $M[2i+1]$ es el hijo derecho de $M[i]$ siempre que $2i+1 \leq n$ con $i \in \mathbb{N}$.

Esta representación de montículo será la utilizada en este documento ya que permite a partir de un nodo acceder con facilidad tanto a su padre como a sus hijos.
Una de las características más relevantes de los montículos es que se pueden restaurar fácilmente, en caso de que se haya modificado el valor de alguna componente. Una aplicación bastante útil que tiene esta estructura es el método de ordenamiento llamado "ordenación por montículos", más conocido como "*Heapsort*".

1.5. COMPLEJIDAD ALGORÍTMICA

Para resolver un problema pueden existir varios algoritmos. La complejidad algorítmica consiste en determinar matemáticamente la cantidad de recursos necesarios por cada uno de los algoritmos, posibilitando seleccionar el más adecuado para resolver el problema.

1.5.1. Instancia

Una instancia de un problema es un conjunto particular de los datos de entrada del problema. Por ejemplo, para encontrar el número mayor entre dos números enteros positivos dados; una instancia de este problema puede ser (324, 79), así como (3, 9), sin embargo (-785,45) no es una instancia pues, -785 es negativo, en otras palabras esta instancia no forma parte del dominio de definición del problema.

1.5.2. Tamaño de una instancia

El tamaño de una instancia formalmente corresponde al número de bits que se requieren para almacenar dicha instancia en un computador utilizando un esquema de codificación definido y razonablemente compacto. En adelante, la palabra tamaño se utilizará para indicar cualquier entero que mida de alguna forma el número de componentes de una instancia y se denotará por $|X|$ el tamaño de la instancia X. Por ejemplo para hablar de grafos, el tamaño de una instancia será el número de nodos o de aristas (o ambos).

1.5.3. Operación elemental

Una operación elemental es aquella cuyo tiempo de ejecución está acotado superiormente por una constante que depende esencialmente de la implementación y la máquina que se utilicen; así, ésta constante no depende ni del tamaño, ni de los parámetros de la instancia considerada.
Como el tiempo exacto requerido por cada operación elemental no es importante, se dice que las operaciones elementales se pueden ejecutar con coste unitario.
En general, se consideran como Operaciones Elementales:

- Operaciones Aritméticas: $+, -, \times, /$, ***div***, ***mod***.
- Operaciones Booleanas: conjunción, disjunción, negación.
- Operaciones de Relación: $=, \neq, <, \leq, >, \geq$.
- Operación de Asignación.

1.5.4. Eficiencia

Se dice que un algoritmo es más eficiente que otro si la cantidad de recursos utilizados para resolver el mismo problema es menor. Los recursos

que se analizan para evaluar la eficiencia de un algoritmo generalmente son el tiempo de ejecución y el espacio de almacenamiento requerido, de los cuales el que se analizará en este trabajo será el tiempo de ejecución, así al referirse a la eficiencia de un algoritmo se estará hablando básicamente de lo rápido que este se ejecuta.

1.5.5. Función de complejidad

Sea X una instancia de un problema y sea A un algoritmo que resuelve el problema. Se denota por t_X el número de operaciones elementales que utiliza el algoritmo A para resolver la instancia X del problema; se define la **función de complejidad** $f : \mathbb{N} \to \mathbb{R}^+$ del algoritmo A como $f(n) = \text{máx}\{t_X : |X| = n\}$.

En la presentación de los algoritmos se utiliza el siguiente esquema, donde t_i es el número de operaciones elementales realizadas en la i-ésima instrucción.

línea	procedimiento	Operaciones elementales(O.E)
	inicio	
1	instrucción 1	t_1
2	instrucción 2	t_2
3	instrucción 3	t_3
$\vdots$	$\vdots$	$\vdots$
k	instrucción k	t_k
	fin	

Con el animo de facilitar el cálculo de la función de complejidad de los algoritmos presentados en este documento, se establecen algunas reglas útiles para el análisis de las estructuras de control que con mayor

frecuencia se encuentran ellos.

1. **Instrucciones Secuenciales**

	(O.E)
procedimiento 1	t_1
procedimiento 2	t_2
procedimiento 3	t_3
$\vdots$	$\vdots$
procedimiento k	t_k

   Si se denota por t_j el número de operaciones elementales del procedimiento j, la función de complejidad de esta secuencia estará dada por,
   $T(n) = t_1 + t_2 + t_3 + \ldots + t_k$; siendo n el tamaño de entrada.

2. **Instrucciones condicionales**

	(O.E)
si (condición) **entonces**	t_c
procedimiento 1	t_1
sino	
procedimiento 2	t_2

   La función de complejidad de una instrucción condicional está dada por:

$$T(n) = t_c + \text{máx}\{t_1, t_2\}$$

3. **Ciclo mientras** (Caso 1)

   Si el número de operaciones elementales de las instrucciones dentro del ciclo no dependen de la variable de control del ciclo

	(O.E)
$i \leftarrow 1$	1
mientras $i \leq n$ **haga**	1
procedimiento	t_1
$i \leftarrow i + 1$	2

La función de complejidad de este ciclo esta dada por:

$$T(n) = (t_1 + 3)n + 2$$

Una forma más general es la siguiente:

	(O.E)
mientras (condición) **haga**	t_c
procedimiento	t_1

La función de complejidad del ciclo ahora es,
$T(n) = (t_c + t_1)N_r + t_c$, donde N_r es el número de veces que se ejecuta el procedimiento.

4. **Ciclo mientras** (caso 2)
Si el número de operaciones elementales de las instrucciones dentro del ciclo dependen de la variable de control del ciclo

	(O.E)
$i \leftarrow 1$	1
mientras $i \leq n$ **(operador lógico)**(condición)**haga**	t_c
procedimiento(i)	$t(i)$
$i \leftarrow i + 1$	2

Para este caso, la función de complejidad del ciclo está dada por:

$$T(n) = t_c(n + 1) + \sum_{i=1}^{n}(t(i) + 2) + 1$$

1.5.6. Notación asintótica

La notación asintótica es utilizada para analizar el comportamiento de una función cuando alguno de sus parámetros tiende a un valor asintótico. Este tipo de análisis puede ser importante para mostrar que dos funciones son aproximadamente iguales entre si, o que el valor de cierta función crece asintóticamente más rápido que el de otra. En el análisis de algoritmos, la notación asintótica se utiliza como un método para medir la eficiencia.

Notación "O grande"(O)

Sea $f : \mathbb{N} \to \mathbb{R}^+$ una función arbitraria. Se denotará por $O(f(n))$ (O grande de $f(n)$) al conjunto de todas las funciones $\hbar : \mathbb{N} \to \mathbb{R}^+$ tales que $f(n) \leq c\, \hbar(n)$ para todo $n \geq n_0$ con $n_0 \in \mathbb{N}$ y para algún escalar $c \in \mathbb{R}^+$. Para indicar que la función g está en el conjunto $O(f(n))$, la notación tradicional es $g = O(f(n))$, y se dirá que g es del **orden de** $f(n)$.

Notación "Omega"(Ω)

Sea $f : \mathbb{N} \to \mathbb{R}^+$ una función arbitraria. Se denotará por $\Omega(f(n))$ (omega de $f(n)$) al conjunto de todas las funciones $\hbar : \mathbb{N} \to \mathbb{R}^+$ tales que $c\, \hbar(n) \leq f(n)$ para todo $n \geq n_0$ con $n_0 \in \mathbb{N}$ y para algún escalar $c \in \mathbb{R}^+$. Igualmente que para la notación O grande, $g = \Omega(f(n))$ indica que la función g está en el conjunto $\Omega(f(n))$, y se lee g es **omega de** $f(n)$.

Notación "Theta"(Θ)

Sea $f : \mathbb{N} \to \mathbb{R}^+$ una función arbitraria. Se representará por $\Theta(f(n))$ (theta de $f(n)$) al conjunto de todas las funciones $\hbar : \mathbb{N} \to \mathbb{R}^+$ tales que $\hbar = O(f(n))$ y $\hbar = \Omega(f(n))$.

Para indicar que $g \in \Theta(f(n))$ se escribirá $g = \Theta(f(n))$ y se dirá que g es de **orden exacto** $f(n)$.

1.6. ORDENACIÓN POR MONTÍCULOS

En la mayoría de los algoritmos voraces, el proceso de selección depende de un previo ordenamiento del conjunto de entrada, por esta razón es necesario introducir algún método de ordenamiento.
Se implementa el método de ordenamiento por *Montículos* ya que éste se ajusta adecuadamente al contexto en el cual se desarrolla el documento y además proporciona algunas herramientas que contribuyen al desarrollo de otras tareas diferentes al ordenamiento. Aunque el método elegido es bastante eficiente, cabe resaltar que se pueden implementar otros como el ordenamiento rápido (*Quicksort*), el cual es similar asintóticamente a *Heapsort* y en la práctica resulta ser más eficiente.

```
    procedimiento intercambiar(var V: vector; x, y: cardinal)   O.E
    var aux: dato
    inicio
1       aux ← V[x]                                              1
2       V[x] ← V[y]                                             1
3       V[y] ← aux                                              1
    fin_intercambiar
```

El procedimiento *intercambiar* como su nombre lo indica, intercambia los elementos de las posiciones x e y de un vector.

Para hallar la función de complejidad de un algoritmo se analiza el número de operaciones elementales (OP) realizadas en cada línea; la siguiente tabla describe paso a paso el cálculo de dicha función. En adelante

para cada algoritmo analizado se utiliza el mismo esquema y método. De esta manera la función de complejidad del procedimiento intercambiar está dada por la siguiente tabla.

líneas	número de operaciones
1-3	3
$f(n) =$	3

Así, $f(n) = O(1)$, donde n es la cantidad de componentes de V.

El procedimiento *hundir*, compara el nodo i-ésimo con sus hijos, en caso de que no se dé la propiedad de orden de del montículo, lo intercambia con el hijo que impida la propiedad; este proceso se repite hasta que el nodo de interés establezca la propiedad de orden del montículo con sus hijos.

```
    procedimiento hundir(var V: vec ; n, i: cardinal)   O.E
    var k, j: cardinal
    inicio
1     k ← i                                              1
      repetir
2       j ← k                                            1
3       si 2j ≤ n y V[2j] > V[k] entonces                5
4         k ← 2j                                         2
        fin_si
5       si 2j < n y V[2j + 1] > V[k] entonces            6
6         k ← 2j + 1                                     3
        fin_si
7        intercambiar(V, j, k)                           3
8     hasta que j = k                                    1
    fin_hundir
```

El análisis de este procedimiento lo presenta la siguiente tabla,

líneas	número de operaciones
3-4	7
5-6	9
2-7	20
2-8	$21\log(n)$
1-8	$21\log(n)+1$
$f(n)=$	$21\log(n)+1$

Por lo tanto la función de complejidad del procedimiento *hundir* es, $f(n) = O(\log(n))$.
Puesto que asintóticamente $\log(n)$ es equivalente a $\lfloor\log(n)\rfloor$, en adelante, en las funciones de complejidad se utiliza $\log(n)$ en lugar de $\lfloor\log(n)\rfloor$.

El procedimiento *flotar* realiza un proceso similar al efectuado por el procedimiento *hundir*, con la diferencia que *flotar* se ocupa de la propiedad de orden del montículo entre el nodo de interés y su padre.

```
    procedimiento flotar(var V: vec; i: cardinal)   O.E
    var k, j: cardinal
    inicio
1     k ← i                                          1
      repetir
2       j ← k                                        1
3       si j > 1 y V[j div 2] < V[k] entonces        4
4         k ← j div 2                                2
        fin_si
5       intercambiar(V, j, k)                        3
6     hasta que j = k                                1
    fin_flotar
```

La siguiente tabla presenta el análisis de este procedimiento

líneas	número de operaciones
3-4	6
2-5	10
2-6	$11\log(n)$
1-6	$11\log(n)+1$
$f(n)=$	$11\log(n)+1$

La función de complejidad $f(n)$ del procedimiento *flotar* está en el orden de $\log(n)$, es decir, $f(n) = O(\log(n))$, donde n es el número de nodos del montículo.

El procedimiento *crear_montículo* como su nombre lo indica, construye un montículo a partir de un vector de n componentes. Esto se logra hundiendo todos los nodos que admitan este procedimiento, o sea, los nodos que tienen altura mayor o igual que uno.

```
    procedimiento crear_montículo(var V: vec; n: cardinal)      O.E
    var i: cardinal
    inicio
1     para i ← (ult2 div 2) bajando hasta 1 hacer               1
2       hundir(V, ult, i)                                       log(n)
      fin_para
    fin_crear_monticulo
```

Sea $f(n)$ la función de complejidad del procedimiento *crear_montículo*. En el peor caso, cuando se tiene un árbol binario completo, es decir, cuando $n = 2^{k+1} - 1$ siendo k es la altura del árbol, el procedimiento *crear_montículo* hunde 2^{k-1} nodos de altura 1, 2^{k-2} nodos de altura 2, y así sucesivamente, finalmente dos(2) nodos de altura $k-1$ y un(1) nodo

(nodo raíz) de altura k así,

$$\begin{aligned}
f(n) &\leq 2 \times 2^{k-1} + 3 \times 2^{k-2} + 4 \times 2^{k-3} + \ldots + (k+1) \times 2^0 \\
f(n) &\leq -2^k + 2^k + 2 \times 2^{k-1} + 3 \times 2^{k-2} + 4 \times 2^{k-3} + \ldots + (k+1) \times 2^0 \\
f(n) &< -2^k + 2^{k+1}(2^{-1} + 2 \times 2^{-2} + 3 \times 2^{-3} + 4 \times 2^{-4} + \ldots) \\
f(n) &< -2^k + 2^{k+1} \sum_{i=1}^{\infty} i \left(\frac{1}{2}\right)^i \\
&= -2^k + 2^{k+1} \left(\frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} \right) \\
&= -2^k + 2^{k+1} 2 \\
&= 2^{k+2} - 2^k \\
&= 2^k(2^2 - 1) \\
&= 3(2^k) < 3n \\
&< 3n
\end{aligned}$$

de esta manera $f(n) = O(n)$.

Después de haber estudiado cada uno de los procedimientos necesarios para el ordenamiento por montículos, finalmente se presenta el procedimiento *ordenar*.

```
    procedimiento ordenar(var V: vec; n: cardinal)                    
    var i: cardinal                                                   
    inicio                                                            
1     crear_montículo(V, n)                                    3n     
2     para i ← n bajando hasta 2 hacer                         1      
3       intercambiar(V, 1, i)                                  3      
4       hundir(V, i − 1, 1)                                    log(n) 
      fin_para                                                        
    fin_ordenar
```

La función de complejidad del procedimiento *ordenar* es exhibida en la siguiente tabla,

líneas	número de operaciones
3-4	$3 + \log(n)$
2-4	$(n-1)(3 + \log(n)) + 1$
1-4	$3n + (n-1)(3 + \log(n)) + 1$
$f(n) =$	$6n + n\log(n) - \log(n) - 2$

De lo anterior se desprende que la función de complejidad del procedimiento *ordenar* $f(n)$ es del orden $n\log(n)$, es decir $f(n) = O(n\log(n))$.

Capítulo 2

ALGORITMOS VORACES

Los algoritmos voraces, ávidos o de avance rápido (en inglés greedy) son utilizados para solucionar problemas de optimización y toman decisiones basándose en la información que tienen de modo inmediato, sin tener en cuenta los efectos que estas decisiones pueden tener a futuro. Por tanto resultan fáciles de diseñar, fáciles de implementar y cuando funcionan son eficientes.

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

- en un grafo ponderado dar la ruta más corta para ir de un nodo a otro
- suponiendo que el sistema monetario de un país está formado por un numero finito de denominaciones, para cualquier cantidad dada, suministrarla en el menor número posible de monedas

En tales contextos, un algoritmo voraz funciona seleccionando la arista o la moneda que parezca más prometedora en un determinado instante, nunca reconsidera su decisión sea cual fuere la situación que pueda surgir más adelante. No hay necesidad de evaluar alternativas, ni de emplear sofisticados procedimientos de seguimiento 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 óptima por el momento.

Hay una segunda función que 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. Normalmente se espera que el problema tenga al menos una solución que sea posible obtener empleando candidatos del conjunto disponible inicialmente.

Hay otra función más, la función de selección, que indica en cualquier momento cual es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados.

Por último, existe una función objetivo que da 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 probable que se seleccione el candidato restante que posea mayor valor individual, como en el problema del cambio de monedas; si por el contrario se intenta minimizar el coste, quizá se seleccione de los candidatos disponibles el de 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 optimice (minimice o maximice, según el caso) el valor de la función objetivo. Los algoritmos voraces avanzan paso a paso, la solución es buscada entre los subconjuntos del conjunto inicial o "Conjunto de Candidatos".

Inicialmente el conjunto de elementos seleccionados es vacío. Entonces, en cada paso se considera adicionar a este conjunto el mejor candidato sin considerar los restantes, siendo guiada esta elección por la función de selección. Una vez seleccionado el nuevo elemento, se verifica si el conjunto con él conformado es 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 el 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 repite hasta que se hayan considerado todos los candidatos.
Cuando el algoritmo voraz funciona correctamente, la primera solución que se encuentre es siempre óptima, es decir que para aplicar el método es necesario que el conjunto de soluciones factibles satisfaga la siguiente propiedad: si un conjunto de candidatos S es una solución factible, entonces cualquier subconjunto S' de S es también una solución factible[1] (esto implica que el conjunto vacío es también una posible solución).

[1]Esta propiedad es llamada principio de optimalidad o propiedad de subestructura óptima.

Estructura General De Un Algoritmo Voraz

Función *Voraz* (C: Conjunto): Conjunto
inicio
  $S \leftarrow \emptyset$ \\ en S se almacenan los elementos de la solución
  mientras ($C \neq \emptyset$) **y no** *solución(S)* **hacer**
    $x \leftarrow seleccionar(C)$
    $C \leftarrow C - \{x\}$
    Si $factible(S \cup \{x\})$ **entonces**
      $S \leftarrow S \cup \{x\}$)
    fin_si
  fin_mientras
  Si *Solución (S)* **entonces devolver** S
  Sino devolver "No hay soluciones"
fin_*Voraz*

Funciones genéricas:

- *Solución*. Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
- *Seleccionar*. Devuelve el elemento 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).

2.1. ALGUNOS PROBLEMAS QUE SE AJUSTAN AL ESQUEMA VORAZ

2.1.1. PROBLEMA DEL CAMBIO EN MONEDAS

Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1, v_2, \ldots, v_n$ donde $v_i \in \mathbb{N}$, para todo i con $1 \leq i \leq n$,

el problema del cambio de dinero consiste en descomponer cualquier cantidad dada M en monedas de ese país utilizando el menor número posible de monedas.

- INSTANCIA: los valores $v_1, v_2, \ldots, v_n$ de cada denominación y la cantidad M a cambiar.
- SOLUCIÓN: un conjunto S de cantidades $c_1, c_2, \ldots, c_n$ de cada de denominación, tal que $\sum_{i=1}^{n} c_i v_i = M$.
- MEDIDA : $m(S) = \sum_{i=1}^{n} c_i$.
- OBJETIVO: Minimizar $m(S)$.

La función *cambio* presentada a continuación, tiene como entradas el valor M a cambiar y un vector V que contiene las denominaciones del sistema monetario; en las primeras cuatro líneas, se inicializa un vector auxiliar que contendrá los valores ordenados y sus respectivas posiciones de entrada, esto se hace para ubicar cada denominacón en el vector de entrada después del ordenamiento. En la línea cinco se ordena el vector antes mencionado en orden decreciente, en la línea seis se inicializa el contador del ciclo, luego en este se recorre el vector auxiliar comenzando en el primer elemento, en cada entrada introduce la cantidad máxima posible de cada denominación considerada; y por último en la línea 14 verifica que el problema se haya resuelto correctamente. Así, la salida de esta función es un vector que contiene las cantidades de cada denominación en el orden de entrada.

Los siguientes son los tipos implementados en el diseño del algoritmo.

Tipo

moneda = **registro**
indice: byte
valor: cardinal
fin_registro

monedas = **registro**
n: cardinal
vec: **vector**[1. .*máx*] de moneda
fin_registro

vecval = **registro**
n: cardinal
vec: **vector**[1. .*máx*] de cardinal
fin_registro

```
    función cambio(var M: cardinal; V: vecval): vecval      O.E
    var     i, d: byte
            aux: monedas
            solución: vecval
    inicio
1       para i ← 1 hasta V.n hacer                          1
2         solución.vec[i] ← 0                               1
3         aux.vec[i].valor ← V.vec[i]                       1
4         aux.vec[i].indice ← i                             1
        fin_para
5       ordenar(aux.vec)                                    n log(n)
6       i ← 1                                               1
7       mientras i ≤ V.n y aux[i].valor ≤ M hacer           3
8         si M = 0 entonces                                 1
9           i ← V.n + 1                                     2
          sino
10          d ← M div aux.vec[i].valor                      2
11          solución.vec[aux.vec[i].indice] ← d             1
12          dec(M, d · aux.vec[i].valor)                    2
13          inc(i)                                          1
          fin_si
        fin_mientras
14      si M > 0 entonces retornar no hay solución          2
15        sino retornar solución                            1
    fin_cambio
```

Sea n el número de denominaciones, la función de complejidad de la función *cambio* viene dada por:

líneas	número de operaciones
2-4	3
1-4	$4n + 1$
1-6	$4n + n\log(n) + 2$
8-13	7
7-13	$10n + 3$
1-15	$14n + n\log(n) + 7$
$f(n) =$	$14n + n\log(n) + 7$

Por lo tanto, $f(n) = O(n\log(n))$.

Aunque la función *cambio* funciona de forma óptima para la mayoría de sistemas monetarios conocidos, se puede encontrar algún sistema y alguna cantidad para los cuales el algoritmo no halla la solución óptima, por ejemplo, si el sistema monetario de un país está conformado por las siguientes denominaciones 1, 6, 13 y se desea cambiar una cantidad 18. El algoritmo retorna un cambio conformado por 1 moneda de valor 13 y 5 de valor 1, para un total de 6 monedas, pero 3 monedas de valor 6 es una mejor solución al problema.

2.1.2. PROBLEMA DE LA MOCHILA

Se dan una mochila de capacidad en peso P y n objetos, en donde el i-ésimo objeto tiene asociado un peso positivo p_i y un valor positivo v_i. El problema consiste en determinar las fracciones $c_1, c_2, \ldots, c_n$ de cada objeto tales que $\sum_{i=1}^{n} c_i p_i \leq P$ y el valor de $\sum_{i=1}^{n} c_i v_i$ sea máximo. Formalmente este problema se define de la siguiente manera:

- INSTANCIA: los valores $v_1, v_2, \ldots, v_n$, los pesos $p_1, p_2, \ldots, p_n$ del los n objetos antes mencionados y la capacidad P de la mochila.

- SOLUCIÓN: un conjunto S de fracciones $c_1, c_2, \ldots, c_n$ de cada objeto, tales que $\sum_{i=1}^{n} c_i p_i \leq P$.
- MEDIDA: $m(S) = \sum_{i=1}^{n} c_i v_i$.
- OBJETIVO: Maximizar $m(S)$.

La función *mochila* tiene como entradas la capacidad P de la mochila y un vector V que almacena las cantidades disponibles y los valores de cada uno de los objetos; entre las líneas 1 y 4, se inicializa un vector auxiliar de tipo objetos, que almacenará en valor los cocientes entre el valor y la cantidad de cada objeto, así como en cantidad las respectivas posiciones de entrada; en la línea cinco se ordena el vector antes mencionado en orden decreciente de valor, en la línea seis se inicializa el contador del ciclo, en dicho ciclo se recorre el vector auxiliar comenzando en el primer elemento; en cada entrada introduce la máxima cantidad posible de cada objeto considerado, teniendo en cuenta la limitación de la capacidad de la mochila. La salida de esta función es un vector que contiene las fracciones de cada objeto que solucionan el problema.

los tipos en este caso son:

Tipo

objeto = **registro**
can: real
val: real
fin_registro

objetos = **registro**
n: cardinal
vec: **vector**[1. .*máx*] de objeto
fin_registro

```
    vecval = registro
      n: cardinal
      vec: vector[1. .máx] de real
    fin_registro

    función mochila(P: real; V: objetos): vecval                    O.E
    var     i: byte
            aux: objetos
            sol: vecval
    inicio
1     para i ← 1 hasta V.n hacer                                      1
2       sol.vec[i] ← 0                                                1
3       aux.vec[i].can ← V.vec[i].val/V.vec[i].can                    2
4       aux.vec[i].val ← i                                            1
      fin_para
5     ordenar(aux.vec)                                                n log(n)
6     i ← 1                                                           1
7     mientras i ≤ V.n hacer                                          1
8       si V.vec[aux.vec[i].val].can < P entonces                     1
9         sol.vec[aux.vec[i].val] ← V.vec[aux.vec[i].val].can         1
10        dec(P, V.vec[aux.vec[i].val].can)                           1
        sino
11        sol.vec[aux.vec[i].val] ← P                                 1
12        i ← n + 1                                                   2
        fin_si
      fin_mientras
13    devolver sol                                                    1
    fin_mochila
```

Sea n el número de objetos, la función de complejidad del algoritmo que soluciona el problema de la mochila será:

líneas	número de operaciones
2-4	4
1-4	$5n+1$
1-6	$5n+n\log(n)+2$
8-12	4
7-12	$5n+1$
1-13	$10n+n\log(n)+4$
$f(n)=$	$10n+n\log(n)+4$

De esta manera se concluye que, $f(n) = O(n\log(n))$.

El siguiente teorema nos garantiza que el algoritmo anteriormente descrito retorna una solución óptima.

TEOREMA 2.1.1. *Si en el problema de la mochila se seleccionan los objetos en orden decreciente de v_i/p_i, el algoritmo de la mochila encuentra una solución óptima.*

Demostración. Supongase, sin perdida de generalidad, que los objetos están ordenados en orden decreciente de valor por unidad de peso, es decir

$$v_1/p_1 \geq v_2/p_2 \geq \ldots \geq v_n/p_n$$

Sea $c_1, c_2, \ldots, c_n$ la solución encontrada por el algoritmo *mochila*. Si $c_i = 1$, para todo i tal que $1 \leq i \leq n$, es claro que esta solución es óptima. De otro modo, supongase que j es el menor índice tal que $c_j < 1$; por la forma como funciona el algoritmo, $c_i = 1$ siempre que $i < j$, $c_i = 0$ para $i > j$ y $\sum_{i=1}^{n} c_i v_i = P$. Ahora, sea $V = \sum_{i=1}^{n} c_i v_i$ el valor total de la solución y $c'_1, c'_2, \ldots, c'_n$ cualquier solución factible, de ahí $\sum_{i=1}^{n} c'_i p_i \leq P$ y por tanto

$\sum_{i=1}^{n}(c_i - c'_i)v_i \geq 0$.

Sea $V' = \sum_{i=1}^{n} c'_i v_i$ el valor de la solución $c'_1, c'_2, \ldots, c'_n$, así

$$V - V' = \sum_{i=1}^{n}(c_i - c'_i)v_i = \sum_{i=1}^{n}(c_i - c'_i)p_i\frac{v_i}{p_i}$$

Notese que cuando $i < j,\ c_i = 1$ y por consiguiente $c_i - c'_i$ es no negativo mientras que $v_i/p_i \geq v_j/p_j$; cuando $i > j,\ c_i = 0$ y por tanto $c_i - c'_i$ es menor o igual a cero, mientras que $v_i/p_i \leq v_j/p_j$ y cuando $i = j,\ v_i/p_i = v_j/p_j$. Por tanto en todos los casos se verifica la desigualdad $(c_i - c'_i)(v_i/p_i) \geq (c_i - c'_i)(v_j/p_j)$, en consecuencia,

$$V - V' \geq (v_j/p_j)\sum_{i=1}^{n}(c_i - c'_i)p_i \geq 0$$

De esta manera se prueba que cualquier solución factible del problema tiene valor no mayor que el de la solución encontrada por la función *mochila*, por lo tanto esta última es óptima. □

2.1.3. PROBLEMA DEL ÁBOL DE RECUBRIMIENTO MÍNIMO

Dado un grafo conexo G 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 S de A con exactamente $|V|-1$ aristas, tal que no contiene ciclos.
- MEDIDA: $m(S) = \sum_{e \in S} p(e)$, siendo $p(e)$ es el peso correspondiente a la arista e.

- OBJETIVO: Minimizar $m(S)$.

El siguiente lema es fundamental en la demostración de la corrección[2] de los algoritmos que se presentarán para resolver este problema.

Lema 2.1.1. *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 aristas 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 el otro en $V - B$. Entonces $S \cup \{e\}$ es prometedor.*

Demostración. Sean G, B y e como en la hipótesis; sea 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 podrían seguir adicionando las demás aristas de U para finalmente tener $S = U$.
En caso contrario, $U \cup \{e\}$ contiene exactamente un ciclo, además debe existir otra arista u que tenga un vértice 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 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 de *Prim*

El algoritmo de *Prim*[3] para resolver este problema empieza a construir el árbol de recubrimiento a partir de un nodo arbitrario de V. Luego, en cada iteración 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

[2]Se refiere al hecho de encontrar una solución óptima.
[3]Este nombre se debe al matemático estadounidense Robert C. Prim(1921) quien publicó este algoritmo en 1957.

cuando el árbol contiene todos los vertices de V.
En el siguiente algoritmo B almacena los vértices del árbol solución y S el conjunto de aristas que lo conforman.

función *prim*$(G = (V, A), p)$: conjunto de aristas
var B: conjunto de vértices
    S: conjunto de aristas
inicio
1 $S \leftarrow \emptyset$
2 $B \leftarrow$ \\ un miembro arbitrario de V
3 **mientras** $B <> V$ **Hacer**
4 *buscar* $e = \{u, v\} \in A$, tal que $u \in B$ y $v \in V - B$
   con $p(e)$ de peso mínimo
5 $S \leftarrow S \cup \{e\}$
6 $B \leftarrow B \cup \{v\}$
   fin_mientras
5 **devolver** S
fin_*prim*

TEOREMA 2.1.2. *El algoritmo de* Prim *descrito anteriormente halla un árbol de recubrimiento mínimo para* G.

Demostración. Esta demostración se hará por inducción matemática sobre el número de aristas del conjunto S. Se demostrará que si el conjunto S es prometedor en algúna instancia del algoritmo, al agregarle una arista adicional éste seguirá siéndolo. Así, después de ser insertadas $n-1$ aristas 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 pueden adicionar las aristas de cualquiera 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 vertices de las aristas de S; ahora como e es la arista de menor peso en A que tiene un único vértice en B por definición, se satisfacen las hipótesis del lema 2.1.1 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 supone que los nodos del grafo están enumerados de 1 a n; se utiliza una matriz de $n \times n$ para representar el grafo de la siguiente manera: $G.mat[i, j]$ contiene el peso de la arista $\{i, j\}$ o ∞[4] 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 *dmínima* tales que *masprox.vec*[i] representa el nodo más cercano al nodo i y *dmínima.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.

Tipo

arista = **registro**
$v1, v2$: cardinal
fin_registro

alturas = **registro**
n: cardinal
vec: **vector**[1. .*máx*] de cardinal
fin_registro

[4]En la práctica, ∞ es un valor mayor que el peso de cualquier arista del grafo.

grafo = **registro**

n: cardinal

mat: **matriz**[1. .$máx$,1. .$máx$] de cardinal

fin_registro

conjuntoaristas = **conjunto** de arista

```
    función prim(G: grafo): conjuntoaristas;
    var i, mínimo: cardinal
        aux: arista
        masprox, dmínima: alturas
        S: conjuntoaristas
    inicio
1       S ← ∅                                              1
2       para i ← 2 hasta G.n hacer                         1
3         masprox.vec[i] ← 1                               1
4         dmínima.vec[i] ← G.mat[i,1]                      1
        fin_para
5       para i ← 1 hasta G.n-1 hacer                       1
6         mínimo ← ∞                                       1
7         para j ← 2 hasta G.n hacer                       1
8           si 0 ≤ dmínima.vec[j] < mínimo entonces        2
9             mínimo ← dmínima.vec[j]                      1
10            k ← j                                        1
            fin_si
          fin_para
11        S ← S ∪ {{masprox.vec[k], k}}                    2
12        dmínima.vec[k] ← −1                              1
13        para j ← 2 hasta G.n hacer                       1
14          si G.mat[j, k] < dmínima.vec[j] entonces       1
15            dminima.vec[k] ← G.mat[j, k]                 1
16            masprox.vec[j] ← k                           1
            fin_si
          fin_para
        fin_para
17      devolver S                                         1
    fin_prim
```

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

líneas	número de operaciones
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	$9n^2+6n-2$
$f(n)=$	$9n^2+6n-2$

Lo anterior indica que la función de complejidad de la función *prim* está en $\mathrm{O}(n^2)$.✓

Algoritmo de *Kruskal*

El algoritmo de *Kruskal*[5] soluciona el problema del árbol de recubrimiento mínimo ordenando, primero, el peso de las aristas 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 produzcan ciclos, para añadir una arista, se evalúa si ésta une dos árboles distintos, de lo contrario no se considera como parte de la solución. Notese que en cada fase el número de árboles del bosque se reduce

[5]En honor al matemático estadounidense Joseph Kruskal(1929) quien publicó este algoritmo en 1956.

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 utilizará los siguientes tipos de datos:

Tipo

arista = **registro**
  $v1, v2$: cardinal
  peso: cardinal
fin_registro

aristas = **registro**
  na: cardinal
  vec: **vector**[1. .*máx*] de arista
fin_registro

Caris = **conjunto** de arista

vcar = **registro**
  n: cardinal
  vec = **vector**[1. .*máx*] de cardinal
fin_registro

Como se mencionó anteriormente, la ejecución de este algoritmo necesita identificar si la arista considerada forma o no un ciclo, para esto se implementará una estructura de conjuntos disjuntos de tal forma que cada nodo está en un único conjunto. Sabiendo que inicialmente el bosque generado por las aristas de la solución contiene n árboles de un solo 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 del nodo i en caso de que $conjunto.vec[i] \neq i$, ó el nodo correspondiente al índice i es la raíz del árbol al cual pertenece, en caso de que $conjunto.vec[i] = i$.
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*.

```
    procedimiento fusionar(var va, vc: vcar; r1, r2: cardinal)   O.E
    var      i: cardinal
    inicio
1       si va.vec[r1] = va[r2] entonces                          1
2          inc(va.vec[r1])                                       1
3           vc.vec[r2] ← r1                                      1
        sino
4          si va.vec[r1] > va.vec[r2] entonces                   1
5             vc.vec[r2] ← vc.vec[r1]                            1
           sino
6             vc.vec[r1] ← vc.vec[r2]                            1
           fin_si
        fin_si
    fin_fusionar
```

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

```
    función buscar(vconj: vcar; k: cardinal): cardinal   O.E
    var      i: cardinal
    inicio
1       i ← k                                            1
2       mientras i ≠ vconj.vec[i] hacer                  1
3          i ← vconj.vec[i]                              1
4       devolver i                                       1
    fin_buscar
```

La función *buscar* realiza en la primera línea una asignación, el ciclo de la línea 2 se ejecuta a lo más tantas veces como la altura del árbol al cual pertenece el nodo k, que como se prueba más adelante es como máximo $\lfloor \log(|vconj|) \rfloor$, 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.

TEOREMA 2.1.3. *Partiendo de la situación inicial "cada casilla del vector contiene su propia posición", un árbol creado mediante una secuencia arbitraria de operaciones* fusionar *tiene como máximo altura* $\lfloor \log(k) \rfloor$, *donde* k *es el número de nodos del árbol.*

Demostración. Esta demostración se hace por inducción matemática sobre el número de nodos del árbol.
Para $k = 1$, claramente la altura de un árbol de un nodo tiene altura 0 y $0 \leq \lfloor \log 1 \rfloor$.
Ahora, sea $k > 1$ y supóngase que se cumple el resultado para todo m con $1 \leq m \leq k$; ahora nótese que un árbol de k nodos únicamente se puede conseguir por la fusión de dos árboles más pequeños que él, supongamos que dichos árboles contienen k_1 y k_2 nodos respectivamente, sin pérdida de generalidad se asume que $k_1 < k_2$, además $k_1 \leq k/2$ y $k_2 \leq k - 1$, ya

que $k = k_1 + k_2$; ahora como $k_1 \geq 1$ y $k_2 \geq 1$ se tiene que $k > a$ y $k > b$. sean h_1 y h_2 las alturas de los árboles más pequeños respectivamente y sea h_k la altura del árbol de k nodos, hay dos posibilidades:
Si $h_1 \neq h_2$, entonces $h_k = \text{máx}(h_1, h_2) \leq \text{máx}(\lfloor \log(k_1) \rfloor, \lfloor \log(k_2) \rfloor)$, haciendo uso de la hipótesis inductiva dos veces, ahora como $k > a$ y $k > b$ entonces, $h_k \leq \lfloor \log(k) \rfloor$.
Si $h_1 = h_2$, en este caso $h_k = h_1 + 1 \leq \lfloor \log(k_1) \rfloor + 1$, usando la hipótesis inductiva sobre k_1. Por otro lado

$$\lfloor \log(k_1) \rfloor \leq \lfloor \log(k/2) \rfloor = \lfloor \log(k) - 1 \rfloor = \lfloor \log(k) \rfloor - 1$$

y por tanto $h_k \leq \lfloor \log(k) \rfloor$. □

La función *kruskal* toma como datos de entrada el conjunto de aristas A del grafo y el número de nodos n; en la primera línea se ordenan las aristas en orden creciente, luego, se inicializan el conjunto *solución* y los vectores *conjunto.vec* y *altura.vec*; el ciclo de la línea 7 recorre las aristas ya ordenadas partiendo de la primera y dentro del ciclo se verifica si la arista considerada forma o no un ciclo (sus vértices se encuentran o no en el mismo conjunto); en caso de que no lo forme, la arista es insertada a la solución y los conjuntos a los cuales pertenecen los vértices son fusionados; de esta manera al salir del ciclo se tendrá un conjunto con $n - 1$ aristas.

```
    función kruskal(var n: cardinal; A: aristas): Caris;          O.E
    var i, u, v, contador: cardinal
        conjunto, altura: alturas
        solución: Caris
    inicio
1       solución ← ∅                                               1
2       contador ← 0                                               1
3       para i ← 1 hasta n hacer                                   1
4         conjunto.vec[i] ← i                                      1
5         altura.vec[i] ← 0                                        1
        fin_para
6       ordenar_por_montículo(A, A.na)                             na log(na)
7       i ← 1                                                      1
8       mientras contador < n − 1 hacer                            2
9         u ← buscar(A.vec[i].v1, conjunto)                        log(n)
10        v ← buscar(A.vec[i].v2, conjunto)                        log(n)
11        si u ≠ v entonces                                        1
12          solución ← solución ∪ {A[i]}                           2
13          inc(contador)                                          1
14          fusionar(altura, conjunto, u, v)                       3
          fin_si
15        inc(i)                                                   1
        fin_mientras
16      retornar solución                                          1
    fin_kruskal
```

La función de complejidad de la función *kruskal*, en términos del número de nodos y de aristas del grafo, está dada por:

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

Es decir, la función de complejidad de *kruskal* está en el $\mathrm{O}(na\log(na))$, pero como se sabe:

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

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

Como se probó anteriormente, los algoritmos de $Prim$ y $Kruskal$ tienen funciones de complejidad que están en el orden exacto n^2 y $na\log n$ respectivamente; luego como $n-1 \leq na \leq n(n-1)/2$ la función de complejidad de *kruskal* acota superiormente a la de *prim*, cuando na es próximo a $(n-1)n/2$ y viceversa cuando na es cercano a $n-1$.

2.1.4. RECORRIDOS DEL CABALLO DE AJEDREZ

Dados un tablero de ajedrez de dimensión $n \times n$ y una casilla inicial, el problema consiste en determinar si es posible recorrer con un caballo todas las casillas del tablero partiendo de la casilla mencionada, sin repetir alguna.

Dado que el problema del recorrido del caballo de ajedrez no es un problema de optimización, no se maneja el mismo esquema planteado para dichos problemas, tan sólo se determina cuáles son las instancias y la solución del problema, de esta forma el problema tendrá:

- INSTANCIA: El tablero, y la casilla inicial de partida.
- SOLUCIÓN: Verdadero o falso.

Para la implementación de este problema, se utiliza una matriz que representa el tablero de ajedrez:

Tipo

tablero = **registro**
  n: cardinal
  mat: **matriz**[1. .$máx$,1. .$máx$] de cardinal
fin_registro

Se utilizará un procedimiento *inictablero* que inicializa la matriz que representa el tablero t asignando 0 a cada casilla, para indicar que no se ha pasado por ninguna de ellas.

```
    procedimiento inictablero(var t: tablero)   O.E
    var      i,j: byte
1       para i ← 1 hasta t.n hacer              1
2          para j ← 1 hasta t.n hacer           1
3             t.vec[i, j] ← 0                   1
           fin_para
       fin_para
    fin_inictablero
```

La siguiente tabla describe el cálculo de la función de complejidad del procedimiento *inictablero*:

líneas	número de operaciones
2-3	$2n+1$
1-3	$2n^2+2n+1$
$f(n)=$	$2n^2+2n+1$

Por tanto el procedimiento *inictablero* efectúa $2n^2+2n+1$ operaciones elementales, es decir su función de complejidad está en $O(n^2)$.

Después de tener las coordenadas de la casilla de partida, es necesaria una función que indique si es posible a partir de una casilla acceder a alguna de sus ocho(8) posibilidades, para ello se utilizará la función *salto* la cual verifica la posibilidad de que desde la casilla (x, y) el caballo salte a la opción i, y además guarda en nx, ny las coordenadas de la posición correspondiente a esta opción; las opciones de movimiento son enumeradas de la forma que muestra la siguiente figura:

	2		3	
1				4
		(x,y)		
8				5
	7		6	

función *salto*(t: tablero; **var** i, x, y, nx, ny: cardinal): boolean O.E
inicio
caso i **de**

1 1: $nx \leftarrow x-2$; $ny \leftarrow y-1$; 5
2 2: $nx \leftarrow x-1$; $ny \leftarrow y-2$; 5
3 3: $nx \leftarrow x+1$; $ny \leftarrow y-2$; 5
4 4: $nx \leftarrow x+2$; $ny \leftarrow y-1$; 5
5 5: $nx \leftarrow x+2$; $ny \leftarrow y+1$; 5
6 6: $nx \leftarrow x+1$; $ny \leftarrow y+2$; 5
7 7: $nx \leftarrow x-1$; $ny \leftarrow y+2$; 5
8 8: $nx \leftarrow x-2$; $ny \leftarrow y+1$; 5

fin_caso
9 **devolver** $1 \leq nx \leq t.n$ **y** $1 \leq ny \leq t.n$ **y** $t.vec[nx,\ ny] = 0$ 8
fin_*salto*

La función *salto* tiene un tiempo de complejidad constante:

líneas	número de operaciones
1-8	12
1-9	20
$f(n) =$	20

En esta función para cualquier entrada se realizan 20 operaciones elementales, es decir que el tiempo de ejecución de salto está en $O(1)$.

En el momento en que el caballo realice un movimiento, es probable que existan varias casillas con posibilidades de salto, la función *cuenta*, como su nombre lo indica retornará la cantidad de casillas a las cuales se tiene acceso desde la casilla (x, y).

```
    función cuenta(t: tablero; x, y: cardinal): cardinal   O.E
    var      acc, i, nx, ny: cardinal
    inicio
1       acc ← 0                                              1
2       para i ← 1 hasta 8 hacer                             1
3          si salto(t, i, x, y, nx, ny) entonces             20
4             inc(acc)                                       1
            fin_si
        fin_para
5       devolver acc                                         1
    fin_cuenta
```

Al igual que la función *salto*, *cuenta* tiene una función de complejidad constante:

líneas	número de operaciones
3-4	21
2-4	177
1-5	179
$f(n) =$	179

Esto es, la función *cuenta* efectúa 179 operaciones elementales en su ejecución, por consiguiente su función de complejidad está en $O(1)$.

Para realizar un nuevo movimiento del caballo se utilizará la función *nuemov*, que recibe el tablero y las coordenadas de la posición actual

del caballo; en su ejecución recorre las opciones desde (x, y) y cuenta las posibilidades de cada una de ellas, retornando finalmente las coordenadas de la opción que tenga más posibilidades; siendo así *nuevomov* la parte voraz del algoritmo (función *Caballo.*) que soluciona el problema.

```
    función nuemov(var t: tablero; x, y: cardinal): booleano   O.E
    var     i,j: byte
            nuevax, nuevay, solx, soly, pos: cardinal
            solucion: booleano
    inicio
1     solucion ← falso                                          1
2     pos ← 9                                                   1
3     para i ← 1 hasta 8 hacer                                  1
4         si salto(t, i, x, y, nuevax, nuevay) entonces         20
5           si cuenta(t, nuevax, nuevay) ≤ pos entonces         180
6             solx ← nuevax                                     1
7             soly ← nuevay                                     1
8             solucion ← verdadero                              1
            fin_si
          fin_si
      fin_para
9     si solucion entonces                                      1
10      x ← solx                                                1
11      y ← soly                                                1
      fin_si
12    devolver solucion                                         1
    fin_nuemov
```

La función de complejidad de la función *nuemov* se ve en la siguiente tabla:

líneas	número de operaciones
5-8	183
4-8	203
3-8	1633
1-12	1639
$f(n) =$	1639

Por lo anterior la función *nuemov* realiza 1639 operaciones elementales, es decir su función de complejidad está en O(1).

Finalmente, la función *Caballo* solucionará el problema utilizando el procedimiento y la funciones anteriores; recibe como datos de entrada el tablero t y las coordenadas de la casilla desde la cual se quiere iniciar el recorrido; en el primer paso lo que hace la función *Caballo* es llenar de ceros la matriz t para indicar que no se ha pasado por ninguna casilla; el ciclo de la línea 3 enumera las casillas que son seleccionadas por la función *nuemov*, en caso de que ésta retorne falso, el caballo ha quedado en una casilla sin opción de continuar avanzando; de esta forma, después de salir del ciclo el tablero tendrá enumeradas sus casillas de acuerdo con el orden en que se han visitado.

```
    función caballo(var t: tablero; x, y: cardinal): booleano   O.E
    var      i: byte
             solución: booleano
    inicio
1       inictablero(t)                                          n^2
2       solución ← verdadero                                    1
3       para i ← 1 hasta t.n^2 hacer                            1
4         t.mat[x, y] ← i                                       1
5         si no(nuemov(t, x, y)) y (i < t.n^2 − 1) entonces     1644
6           solución← falso                                     1
            salir
          fin_si
        fin_para
7       retornarsolución                                        1
    fin_caballo
```

La siguiente tabla, relaciona las operaciones elementales realizadas por la función *caballo*, para calcular su función de complejidad:

líneas	número de operaciones
5-6	1645
4-6	1646
3-6	$1647n^2 + 1$
1-7	$1648n^2 + 3$
$f(n) =$	$1648n^2 + 3$

De esta manera, la función de complejidad de la función *caballo* es $f(n) = 1648n^2 + 3 = O(n^2)$.

El problema de los recorridos del caballo de ajedrez no siempre se puede solucionar por medio del algoritmo *caballo*. Por ejemplo, para $n = 5$, $x_0 = 5$ e $y_0 = 3$ la función *caballo* retorna falso a pesar de que el problema tiene solución para esta instancia. Sin embargo, sí la encuentra para $n = 5$, $x_0 = 1$ e $y_0 = 3$, para $n = 5$, $x_0 = 3$ e $y_0 = 1$ y para $n = 5$, $x_0 = 3$ e $y_0 = 5$.

2.1.5. CÓDIGOS 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 está definido de la siguiente manera:
Dados n caracteres y sus respectivas frecuencias en un texto, el problema de los *códigos de Huffman*[6] consiste en determinar un código binario[7] para representar cada uno de los caracteres, de tal manera que el número de bits requeridos para representar el texto sea mínimo. Este problema se define formalmente de la siguiente manera:

- INSTANCIA: los caracteres $c_1, c_2, \ldots, c_n$ y sus frecuencias $f_1, f_2, \ldots, f_3$.
- SOLUCIÓN: Un conjunto S de códigos binarios $cod(c_1), cod(c_2), \ldots$,$cod(c_n)$.
- MEDIDA: $m(S) = \sum_{i=1}^{n} f_i |cod(c_i)|$, donde $|cod(c_i)|$ es la longitud[8]del código binario $cod(c_i)$.

[6]En honor al ingeniero eléctrico estadounidense David A. Huffman(1925-1999) quien publicó este algoritmo en 1953.

[7]Código binario es una sucesión finita de bits, utilizada para almacenar información en un computador.

[8]Cantidad de bits de un código binario.

- OBJETIVO: Minimizar $m(S)$.

La solución que se encuentra está representada por un árbol binario[9] 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 caracter c.

Tipo

pcar = $\uparrow$ *car*

Este es el tipo puntero, el cual almacena la dirección de memoria de una variable de tipo car.

car = **registro**
$frec$: cardinal
hd, hi:pcar
fin_registro

cars = **registro**
n: cardinal
vec: **vector**[1. .$máx$] de pcar
fin_registro

[9] Árbol de códificación óptima.

vecval = **registro**
  n: cardinal
  vec: **vector**[1. .$máx$] de cardinal
fin_registro

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

	función *extraermin*(**var** C: cars): pcar	O.E
	var *i, menor*: cardinal	
	inicio	
1	*extraermin* $\leftarrow C.vec[1]$	1
2	*intercambiar*$(C.vec, 1, C.n)$	3
3	*dec(C.n)*	1
4	*hundir*$(C.vec, C.n, 1)$	$\log(n)$
	fin_*extraermin*	

El análisis de la función *extraermin* está representado por la siguiente tabla,

líneas	número de operaciones
1-4	$\log(n) + 5$
$f(n) =$	$\log(n) + 5$

Por consiguiente, la función de complejidad de la función *extraermin* está en $O(\log(n))$.

La función *insertar* introduce el elemento x al final de C, y posteriormente lo hace flotar para mantener la propiedad de montículo,

	procedimiento *insertar*(**var** C: cars; x: pcar)	O.E
	inicio	
1	$inc(C.n)$	1
2	$C.vec[C.n] \leftarrow x$	1
3	$flotar(C.vec, C.n)$	$\log(n)$
	fin_*insertar*	

El procedimiento *insertar* presenta la función de complejidad $f(n)$ descrita por la siguiente tabla,

líneas	número de operaciones
1-4	$\log(n) + 2$
$f(n) =$	$\log(n) + 2$

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

```
    función Huffman(C: veccar): pcar                    O.E
    var i: cardinal; aux: pcar;
    inicio
1       crearmontículo_de_mínimos(C.vec)                3n
2       para i ← 1 hasta C.n-1 hacer                    1
3         nuevo(aux)                                    1
          con aux ↑ hacer
4           hi ← extraermin(C)                          log(n)
5           hd ← extraermin(C)                          log(n)
6           frec ← hi ↑ .frec + hd ↑ .frec              2
          fin_con
7         insertar(C, aux)                              log(n)
        fin_para
8       devolver C.vec[1]                               1
    fin_huffman
```

Su análisis de complejidad es presentado a continuación,

líneas	número de operaciones
5-8	$3\log(n) + 4$
4-8	$3n\log(n) - 3\log(n) - 3$
1-8	$3n\log(n) - 3\log(n) + 3n + 1$
$f(n) =$	$3n\log(n) - 3\log(n) + 3n + 1$

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

A continuación se prueba la corrección del algoritmo antes descrito.

Lema 2.1.2. *Sea C un conjunto de caracteres, en el cual cada caracter c tiene asociada una frecuencia f_c. Sean x e y dos caracteres en C con las más bajas frecuencias. Entonces, existe un árbol de códificación óptima para C en el cual los códigos para x e y tienen la misma longitud y difieren tan sólo en el último bit.*

Demostración. Sean T un árbol binario de codificación óptima para C, a y b hojas de profundidad máxima hijas de un mismo nodo en T y x e y los caracteres de C con las frecuencias más bajas.
Se probará que T, que es un árbol óptimo, se puede transformar en otro árbol también óptimo, en el que los caracteres x e y sean hojas de profundidad máxima hijas del mismo nodo.
Supóngase sin pérdida de generalidad que $f_a \leq f_b$ y $f_x \leq f_y$, entonces $f_x \leq f_a$ y $f_y \leq f_b$ por ser x e y los caracteres de menor frecuencia.
Sea T' el árbol que resulta al intercambiar en T la posición que ocupan las hojas a y x y sea $|cod_T(c)|$ la profundidad del caracter c en el árbol T. Entonces la diferencia entre el peso total $m(T)$ de la codificación representada por el árbol T y el peso total $m(T')$ de la codificación exhibida por T' es:

$$\begin{aligned} m(T) - m(T') &= \sum_{c \in C} f_c |cod_T(c)| - \sum_{c \in C} f_c |cod_{T'}(c)| \\ &= f_x |cod_T(x)| + f_a |cod_T(a)| - f_x |cod_{T'}(x)| - f_a |cod_{T'}(a)| \\ &= f_x |cod_T(x)| + f_a |cod_T(a)| - f_x |cod_T(a)| - f_a |cod_T(x)| \\ &= (f_a - f_x)(|cod_T(a)| - |cod_T(x)|) \\ &\geq 0 \end{aligned}$$

puesto que, $f_c|cod_T(c)| - f_c|cod_{T'}(c)| = 0$, para $c \in C - \{a, x\}$; además $|cod_T(x)| = |cod_{T'}(a)|$ y $|cod_{T'}(x)| = |cod_T(a)|$, por último $f_a \geq f_x$ y $|cod_T(a)| \geq |cod_T(x)|$.

De forma similar se puede construir el árbol T'' intercambiando b e y en T'. Con este intercambio se tiene que $m(T') - m(T'') \geq 0$. Por tanto $m(T'') \leq m(T)$ y como T es óptimo $m(T) \leq m(T'')$, lo cual implica que $m(T'') = m(T)$; así T'' es un árbol óptimo en el cual x e y aparecen como hojas de máxima profundidad hijas del mismo nodo. □

El lema anterior implica que el proceso de construcción de un árbol óptimo puede iniciar con la selección voraz "fusionar los caracteres de menor frecuencia".

El siguiente lema muestra que el algoritmo de $Huffman$ posee la propiedad de subesructura óptima.

Lema 2.1.3. *Sea C un conjunto de caracteres con sus respectivas frecuencias y T un árbol binario que representa una codificación óptima para C. Sean x e y dos hojas hijas de un mismo nodo z en T, entonces si se considera z como un caracter con frecuencia $f_z = f_x + f_y$, entonces el árbol $T' = T - \{x, y\}$ representa un código óptimo para el conjunto de caracteres $C' = C - \{x, y\} \bigcup \{z\}$.*

Demostración. Sea $c \in C - \{x, y\}$ un caracter arbitrario, de ahí resulta que $|cod_T(c)| = |cod_{T'}(c)|$; luego $f_c|cod_T(c)| = f_c|cod_{T'}(c)|$. Como $|cod_T(x)| = |cod_T(y)| = |cod_{T'}(z)| + 1$ se tiene que,

$$\begin{aligned} f_x|cod_T(x)| + f_y|cod_T(y)| &= (f_x + f_y)(|cod_{T'}(z)| + 1) \\ &= f_z|cod_{T'}(z)| + (f_x + x_y) \end{aligned}$$

Por lo que $m(T) = m(T') + f_x + f_y$.

Si T' representa una codificación no óptima de C', entonces existe un árbol T'' cuyas hojas son caracteres de C' tal que $m(T'') < m(T')$. Puesto

que z es tratado como un caracter en C', aparece como una hoja en T''. Si se adicionan x e y como hijos de z entonces se obtiene una códificación óptima de C tal que $m(T'') + f_x + f_y < m(T)$ lo cual contradice la optimalidad de T. Por tanto T' debe ser o'ptimo para C'. □

TEOREMA 2.1.4. *La función $Huffman$ retorna un árbol que representa una codificación óptima para cualquier instancia.*

Demostración. Este resultado es consecuencia inmediata de los lemas 2.1.2 y 2.1.3. □

2.1.6. EJEMPLO DE LA EFICIENCIA DE LOS ALGORITMOS VORACES CON RESPECTO OTROS

La función *cambio*2 que se presenta a continuación emplea la técnica "programación dinámica"para resolver correctamente el problema del cambio en monedas, en esta función se utilizan los siguientes tipos de datos:

Tipo

matriznxM = **registro**
  n, M: byte
  mat: **matriz**[1. .*máx*,1. .*máx*] de cardinal
fin_registro

veccar = **registro**
  n: cardinal
  vec: **vector**[1. .*máx*] de cardinal
fin_registro

```
    función cambio2(C: veccar; M: matriznxM): veccar                    O.E
    var i, j: cardinal; solucion: veccar;
    inicio
1     para i ← 1 hasta C.n hacer                                          1
2       M.mat[i, 0] ← 0                                                   1
3       solucion.vec[i] ← 0                                               1
      fin_para
4     para i ← 1 hasta C.n hacer                                          1
5       para j ← 1 hasta Cantidad.M hacer                                 1
6         si i = 1 y j < C.vec[i] entonces                                3
7           M.mat[i, j] ← M + 1                                           2
          sino
8           si i = 1 entonces M.mat[i, j] ← 1+M.mat[1, j−C.vec[1]]        4
            sino
9             si j < C.vec[i] entonces M.mat[i, j] ← 1+M.mat[i−1, j]      4
              sino
10              M.mat[i, j] ← mín{M.mat[i−1, j], 1+M.mat[i, j−C.vec[i]]}  5
              fin_si
            fin_si
          fin_si
        fin_para
      fin_para
11    dec(i)                                                              1
12    dec(j)                                                              1
13    mientras i > 0 o j > 0 hacer                                        3
14    si M.mat[i, j] = M.mat[i−1, j] entonces dec(i)                      3
        sino
15      si M.mat[i, j] = 1 + M.mat[i, j−C.vec[i]] entonces                3
16        j ← j−C.vec[i]                                                  2
17        inc(solucion.vec[i])                                            1
        sino
18        i ← 0                                                           1
19        j ← 0                                                           1
        fin_si
      fin_si
20    devolver solucion                                                   1
    fin_cambio2
```

La función de complejidad del anterior algoritmo lo presenta la siguiente

tabla:

líneas	número de operaciones
1-3	$3n + 1$
6-10	10
5-10	$11M + 1$
4-10	$11Mn + n + 1$
1-12	$11Mn + 4n + 4$
14-19	8
13-19	$11n + 3$
1-20	$11Mn + 15n + 8$
$f(n, M) =$	$11Mn + 15n + 8$

De esta manera la función de complejidad de *cambio*2 depende tanto de la cantidad de denominaciones como del valor a cambiar y está en el orden exacto nM. Nótese que para valores de M superiores a $\log(n)$ la función de complejidad de la función *cambio*2 supera en número operaciones elementales a la función de complejidad de la función *cambio* evidenciando la eficiencia del algoritmo voraz *cambio* con respecto a la función *cambio*2. Por lo tanto la función *cambio* aunque no resuelve el problema del cambio en monedas para todas las posibles instancias, retorna soluciones óptimas con gran eficiencia en un número considerable de casos.

Capítulo 3

TEORÍA DE MATROID

Formalmente la teoría de matroids es usada para demostrar la corrección de un algoritmo voraz, es decir para probar que un algoritmo voraz proporciona soluciones óptimas a cierto problema; aunque esta teoría no abarca la totalidad de los casos en que se utilizan los algoritmos voraces, es de gran utilidad en ciertos problemas comunes en el estudio de este tipo de algoritmos.

3.1. MATROIDS

Una **matroid** es un par ordenado $M = (S, \mathcal{I})$ que satisface las siguientes condiciones.

1. S es un conjunto finito no vacío.

2. $\mathcal{I}$ es una familia no vacía de subconjuntos de S, tal que si $B \in \mathcal{I}$ y $A \subseteq B$, entonces $A \in \mathcal{I}$. Se dice que $\mathcal{I}$ es **hereditaria** si satisface la anterior propiedad. $\mathcal{I}$ es llamada familia de subconjuntos **independientes** de S, nótese que el conjunto vacío $\emptyset$ es un miembro de $\mathcal{I}$.

3. Si $A \in \mathcal{I}$, $B \in \mathcal{I}$, A y B finitos con $|A| < |B|$, donde $|A|$ y $|B|$ es el cardinal de A y B respectivamente, entonces hay algún elemento

$x \in B - A$ tal que $A \cup \{x\} \in \mathcal{I}$. Se dice que M satisface la propiedad de cambio.

La palabra "matroid"se debe a Hassler Whitney. Él estudió las matroids matriciales, en las cuales los elementos de S son las filas de una matriz dada y un conjunto de filas es independiente si las filas son linealmente independientes en el sentido usual. Es fácil probar que esta estructura define una matroid.

Ejemplo 3.1.1. *$M = (S, \mathcal{I})$ con $S = \{1, 2, 3, 4, 5, 6\}$ e $\mathcal{I} = \{\{1\}, \{3\}, \{5\}, \{1, 3\}, \{3, 5\}, \{1, 5\}, \{1, 3, 5\}\}$ es una matroid.*

Ejemplo 3.1.2. *$M = (S, \mathcal{P}(S))$ es una matroid, donde S es cualquier conjunto finito no vacío.*

TEOREMA 3.1.1. *Sea S cualquier conjunto finito e $\mathcal{I}_k$ la colección de todos los subconjuntos de S, tal que $|K| \leq k$, para todo $K \in \mathcal{I}_k$, con $k \leq |S|$ y $k \in \mathbb{N}$, entonces $M = (S, \mathcal{I}_k)$ es una matroid.*

Demostración. Claramente, S es no vacío.
Por otro lado si $B \in \mathcal{I}$ y $A \subseteq B$ entonces $|A| \leq |B|$ luego, por hipótesis $|B| \leq k$ lo que implica que $|A| \leq k$, es decir $A \in \mathcal{I}$. Así, $\mathcal{I}$ es hereditaria. finalmente, si $A, B \in \mathcal{I}$ y $|A| < |B|$ entonces, $B - A \neq \emptyset$; ahora, sea $x \in B - A$ arbitrario, como $B \leq k$, se tiene que $|A| < k$ y por tanto $|A \cup \{x\}| \leq k$; así $A \cup \{x\} \in \mathcal{I}$ y M es una matroid. □

TEOREMA 3.1.2. *Dada una matriz con valores reales T de dimensión $n \times n$. $M = (S, \mathcal{I})$ es una matroid; donde S es el conjunto de columnas de T y $A \in \mathcal{I}$ si y sólo si las columnas de A son linealmente independientes.*

Demostración. Es claro que S es diferente de vacío.
Se probará ahora que $\mathcal{I}$ es hereditaria. Sea $B \in \mathcal{I}$ y $A \subseteq B$, como las columnas de A están en B y en B éstas son linealmente independientes, las columnas de A son linealmente independientes, por tanto $A \in \mathcal{I}$.
Veamos que M satisface la propiedad de cambio. Sea $A \in \mathcal{I}$ y $B \in \mathcal{I}$ con $|A| < |B|$, se probará que existe $x \in B - A$ tal que $A \cup \{x\} \in \mathcal{I}$; por contradicción, supóngase que para todo $x \in B$, $A \cup \{x\} \notin \mathcal{I}$, es decir toda columna de B se puede expresar como combinación lineal de las columnas de A. Como A es linealmente independiente, el conjunto $sg\{A\}$ de todas las combinaciones lineales de las columnas de A es un espacio vectorial cuya base es A, luego por la suposición hecha $B \subseteq sg\{A\}$ y como $|B| > |A|$, B es linealmente dependiente en $sg\{A\}$; en consecuencia B es linealmente dependiente en $\mathbb{R}^n$ lo cual contradice $B \in \mathcal{I}$. Por lo tanto M es una matroid. □

TEOREMA 3.1.3. *Si $G = (V, A)$ es un grafo no dirigido y no trivial, entonces $M_G = (S_G, \mathcal{I}_G)$ es una matroid, donde S_G es el conjunto de aristas A de G e $\mathcal{I}_G$ la familia de subconjuntos acíclicos de A.*

Demostración. Claramente $S_G = A$ es un conjunto finito y no vacío. Además, $\mathcal{I}_G$ es hereditario ya que un subconjunto de un bosque acíclico es un bosque acíclico; poniéndolo en otros términos, al quitar aristas de un conjunto de aristas acíclico no se pueden crear ciclos. Así, resta probar que M_G satisface la propiedad de intercambio. Supóngase que A y B son bosques de G y que $|B| > |A|$. Es decir, A y B son conjuntos acíclicos de aristas y B contiene mas aristas que A. como se sabe, un bosque con k aristas contiene $|V| - k$ árboles, siendo V el conjunto de vértices de G (Para ilustrar esto, inicialmente se tienen $|V|$ árboles en el bosque ya que no se han insertado aristas. Luego, por cada arista adicionada al

bosque se reduce el número de árboles en uno). Así, el bosque A contiene $|V| - |A|$ árboles y el bosque B contiene $|V| - |B|$ árboles. Ya que B tiene menos árboles que A y en ambos bosques se encuentran los mismos vértices, el bosque B debe contener algún árbol T cuyos vértices están en al menos dos árboles diferentes de A. Además, dado que T es conexo, éste debe contener una arista $\{u, v\}$ tal que los vértices u y v están en árboles diferentes en el bosque A; puesto que la arista $\{u, v\}$ conecta dos árboles diferentes en el bosque A, la arista $\{u, v\}$ puede ser adicionada al bosque A sin crear un ciclo. Por lo tanto, $A \cup \{\{u, v\}\} \in \mathcal{I}$ y en consecuencia M_G es una matroid. □

Dada una matroid $M = (S, \mathcal{I})$, a un elemento $x \notin A$ se le llama **extensión** de $A \in \mathcal{I}$, si x puede ser adicionado a A preservando independencia, esto es, x es una extensión de A, si $A \cup \{x\} \in \mathcal{I}$. Como ejemplo, considerese la matroid M_G dada por el grafo G. Si A es un conjunto independiente de aristas, entonces, la arista e es una extensión de A si, y sólo si e no está en A y la adición de e no crea un ciclo. Si A es un subconjunto independiente de una matroid M, se dice que A es **maximal** si no tiene extensiones, es decir A es maximal si no está contenido en algún subconjunto independiente de M más grande. La siguiente propiedad es usada frecuentemente.

TEOREMA 3.1.4. *Todo subconjunto independiente maximal en una matroid M tiene el mismo tamaño.*

Demostración. Supongase lo contrario, que A es un subconjunto independiente maximal de M y que existe otro subconjunto independiente maximal B de M más grande. Entonces, la propiedad de intercambio implica que A es extendible a un conjunto independiente más grande

$A \cup \{x\}$ para algún $x \in B - A$, contradiciendo la suposición de que A es maximal. □

Como ilustración de este teorema, considérese una matroid M_G dada por un grafo conexo no dirigido $G = (V, A)$. Cada subconjunto maximal independiente de M_G debe ser un árbol sin costo con exactamente $|V| - 1$ aristas que conectan todos los vértices de G. Tal árbol es llamado **árbol de recubrimiento** de G.
Se dice que una matroid $M = (S, \mathcal{I})$ es **ponderada** si existe una función $\omega : S \to \mathbb{R}^+$, que asigna un peso estrictamente positivo $\omega(x)$ a cada elemento $x \in S$. La función de peso extendida a subconjuntos de S es:

$$\omega(A) = \sum_{x \in A} \omega(x) \text{ para cualquier } A \subseteq S.$$

Por ejemplo, si se denota por $\omega(e)$ la longitud de una arista e en una matroid M_G dada por el grafo G, entonces ω(A) es la longitud total de la aristas en el conjunto de aristas A.

3.2. ALGORITMOS VORACES SOBRE UNA MATROID PONDERADA

Muchos problemas para los cuales una aproximación voraz provee soluciones óptimas pueden ser formulados en términos de búsqueda de un subconjunto independiente de peso máximo en una matroid ponderada. Es decir, se está dando una matroid ponderada $M = (S, \mathcal{I})$ con función de peso ω y se quiere encontrar un conjunto independiente $A \in \mathcal{I}$ tal que $\omega(A)$ sea máximo. Tal conjunto que es independiente y de peso máximo será llamado subconjunto **óptimo** de la matroid. Puesto que el peso

$\omega(x)$ para cualquier elemento $x \in S$ es siempre positivo, un subconjunto óptimo es siempre un subconjunto independiente maximal, esto siempre ayuda a crear A tan grande como sea posible.
Por ejemplo, en el problema del árbol de recubrimiento mínimo, se cuenta con un grafo conexo no dirigido y no trivial $G = (V, E)$ y una función de longitud ω tal que $\omega(e)$ es la longitud (positiva) de la arista e.(Se puede usar el término "longitud"para referirse al peso original de la arista para el grafo, se reserva el término "peso" para hacer referencia al peso en la matroid asociada) Se pide encontrar un subconjunto de aristas que conecten todos los vértices del grafo y además que tenga longitud total mínima. Para ver esto como un problema de encontrar un subconjunto óptimo en una matroid, considerese la matroid ponderada M_G con función de peso ω' donde $\omega'(e) = \omega_0 - \omega(e)$ y ω_0 es más grande que la longitud máxima de cualquier arista. En esta matroid ponderada, todos los pesos son positivos y un subconjunto óptimo es un árbol de recubrimiento con peso total mínimo en el grafo original. Más específicamente, cada subconjunto independiente maximal A corresponde a un árbol de recubrimiento y ya que $\omega'(A) = (|V| - 1)\omega_0 - \omega(A)$ para cualquier subconjunto independiente maximal A, el subconjunto independiente que maximiza $\omega'(A)$ debe minimizar $\omega(A)$. Así, cualquier algoritmo que pueda encontrar un subconjunto óptimo en una matroid arbitraria puede resolver el problema del árbol de recubrimiento mínimo.

Aquí se dá un algoritmo que trabaja para cualquier matroid ponderada. El algoritmo toma como entrada una matroid $M = (S, \mathcal{I})$ con una función de peso positivo asociado ω y retorna un subconjunto óptimo A. En seudocódigo se denotan las componentes de M por $S[M]$ e $\mathcal{I}[M]$ y la función de peso por ω. El algoritmo es voraz porque considera cada elemento $x \in S$ a su vez en orden decreciente de peso e inmediatamente

lo adiciona al conjunto A si $A \cup \{x\}$ es independiente.

función $VORAZ(M, \omega)$
inicio
1 $A \leftarrow \emptyset$
2 Ordena $S[M]$ en orden decreciente de peso por ω
3 **Para** cada $x \in S[M]$, tomado en orden decreciente de peso por $\omega(x)$
4 **Hacer si** $A \cup \{x\} \in \mathcal{I}$ **entonces** $A \leftarrow A \cup \{x\}$
fin_si
fin_para
5 **devolver** A
fin_*VORAZ*

Los elementos de S son considerados a su vez, en orden decreciente de peso. Si el elemento x inicialmente considerado puede agregarse a A mientras A se mantiene independiente, es insertado a la solución; de otro modo, x es descartado. Puesto que vacío es independiente por la definición de matroid y ya que x es adicionado únicamente a A si $A \cup \{x\}$ es independiente, el subconjunto A es siempre independiente, por inducción. Por consiguiente $VORAZ$ siempre retorna un subconjunto independiente A. Se verá en un momento que A es un subconjunto de máximo peso posible, así que A es un subconjunto óptimo. El tiempo de ejecución de $VORAZ$ es fácil de analizar. Sea $n = |S|$, la fase de ordenamiento de $VORAZ$ toma tiempo $\mathrm{O}(n\ \log(n))$. La línea 4 es ejecutada exactamente n veces, una vez por cada elemento $x \in S$. Cada ejecución de la línea 4 requiere un chequeo sobre la independencia de $A \cup \{x\}$. Si tal chequeo toma tiempo $\mathrm{O}(f\ (n))$, la ejecución del algoritmo entero toma tiempo $\mathrm{O}(n \log(n) + nf\ (n))$. A continuación se prueba que la función $VORAZ$ retorna un subconjunto óptimo.

Lema 3.2.1 (Las matroids exhiben la propiedad de selección voraz). *Supongase que $M = (S, \mathcal{I})$ es una matroid con peso y función de peso ω y que S está ordenado en orden decreciente de peso. Sea x el primer elemento de S tal que $\{x\}$ es independiente, si existe, entonces hay un subconjunto óptimo A de S que contiene a x.*

Demostración. Sea B cualquier subconjunto óptimo no vacío, se asume que $x \notin B$, de lo contrario se toma $A = B$ y la prueba termina. Ahora, ningún elemento de B tiene peso mayor que $\omega(x)$; puesto que $y \in B$ implica que $\{y\}$ es independiente, pues $B \in \mathcal{I}$ e $\mathcal{I}$ es hereditaria. Así, la selección de x nos garantiza que $\omega(x) \geq \omega(y)$ para $y \in B$. Se construye el conjunto A a partir de $\{x\}$. Por la selección de x, A es independiente y aplicando la propiedad de intercambio se encuentra un nuevo elemento de $B - A$ que puede ser adicionado a A preservando su independencia; al realizar este proceso repetidamente hasta que $|A| = |B|$ se tiene que $A = (B - \{y\}) \cup \{x\}$ para algún $y \in B$, y así

$$\begin{aligned} \omega(A) &= \omega(B) - \omega(y) + \omega(x) \\ &\geq \omega(B) \end{aligned}$$

ya que B es óptimo, A también debe ser óptimo y como $x \in A$, el lema queda probado. □

A continuación se prueba que si un elemento nos es una opción inicialmente, entonces no puede serlo después.

Lema 3.2.2. *Sea $M = (S, \mathcal{I})$ una matroid. Si x es un elemento de S tal que x no es una extensión de vacío, entonces x no es una extensión de cualquier subconjunto independiente A de S.*

Demostración. La prueba es por contradicción. Se Asume que x es una extensión de A pero no de vacío; puesto que x es una extensión de A, se tiene que $A \cup \{x\}$ es independiente. Ya que $\mathcal{I}$ es hereditaria, $\{x\}$ debe ser independiente lo cual contradice la suposición de que x no es extensión de vacío. □

El lema 3.2.2 dice que cualquier elemento que no pueda ser usado inmediatamente nunca puede ser usado. Por consiguiente, *VORAZ* no puede generar un error por haber pasado por alto cualquier elemento inicial de S que no sea una extensión de vacío, ya que ellos nunca serán usados.

Lema 3.2.3 (Las matroids exhiben la propiedad de subestructura óptima). *Sea x el primer elemento seleccionado por* VORAZ *para la matroid $M = (S, \mathcal{I})$. El problema de encontrar un subconjunto independiente de peso máximo se reduce a encontrar un subconjunto independiente de peso máximo en la matroid $M' = (S', \mathcal{I}')$, donde:*

- $S' = \{y \in S : \{x, y\} \in \mathcal{I}\ \}$.
- $\mathcal{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\ \}$ *y la función de peso de M' es la función de peso de M, restringida a S'. (M' es llamada la* ***contracción*** *de M por el elemento x).*

Demostración. Si A es cualquier subconjunto independiente de peso máximo de M que contiene a x, entonces $A' = A - \{x\}$ es un subconjunto independiente de M', inversamente, cualquier subconjunto independiente A' de M' produce un subconjunto independiente $A = A' \cup \{x\}$ de M. Dado que en ambos casos se tiene $\omega(A) = \omega(A') + \omega(x)$, una solución de peso máximo que contiene a x en M produce una solución de peso máximo en M' y viceversa. □

TEOREMA 3.2.1 (Corrección de los algoritmos voraces sobre las matroids). *Si $M = (S, \mathcal{I})$ es una matroid con peso y función de peso ω, entonces al llamar* VORAZ*(M, ω) retorna un subconjunto óptimo.*

Demostración. Por el lema 3.2.2 cualquier elemento que se haya pasado por alto inicialmente por no ser extensión de vacío puede ser olvidado, ya que nunca puede ser usado. Una vez el primer elemento x es seleccionado, el lema 3.2.1 implica que $VORAZ$ no errará por adicionar x a A, dado que existe un subconjunto óptimo que contiene a x. Finalmente, el lema 3.2.3 implica que el problema restante es encontrar un subconjunto óptimo en la matroid M' que es una contracción de M por x. Después, el procedimiento $VORAZ$ asigna $\{x\}$ a A, todos los pasos restantes pueden ser interpretados como actuando en la matroid M' ya que B es independiente en M' si y solo si $B \cup \{x\}$ es independiente en M, para todo subconjunto $B \in \mathcal{I}'$, así, la posterior operación de $VORAZ$ es encontrar un subconjunto independiente de peso máximo para M' y la operación final de $VORAZ$ es encontrar un subconjunto independiente de peso máximo para M. □

Este último resultado es de gran utilidad ya que garantiza la corrección de aquellos algoritmos que posean la misma estructura del algoritmo $VORAZ$ ilustrado anteriormente.
Para aplicar este resultado a un algoritmo que dice resolver algún problema de optimización, se debe establecer semejanza entre la estructura del algoritmo y la de $VORAZ$, además de asociar una matroid al problema que dice resolver. De esta manera se nota que:

- No es posible aplicar este resultado al problema del cambio de monedas, ya que el algoritmo voraz que lo resuelve no siempre encuentra

una solución óptima.

- No es posible aplicar la teoría de matroid a los problemas de la mochila, del cambio de monedas y de los recorridos del caballo de ajedrez.

- A pesar de que el problema de la mochila siempre es resuelto de forma óptima por medio de un algoritmo voraz, no es posible asociar una matroid al problema, puesto que el conjunto de soluciones factibles del problema es infinito.

- Como se probó en este capítulo, un grafo conexo no dirigido se puede asociar a una matroid y como fácilmente se puede notar, la estructura del algoritmo $kruskal$ se asemeja a la de $Voraz$; así, el anterior teorema es aplicable a $Kruskal$ en el problema del árbol de recubrimiento mínimo. Por otro lado, como la estructura del algoritmo de $Prim$ difiere considerablemente con la de $VORAZ$ ya que $Prim$ no ordena inicialmente el conjunto de aristas, no es posible aplicar el anterior resultado.

- El problema de los recorridos del caballo de ajedrez no es un problema de optimización, y por consiguiente no es aplicable esta teoría.

- No se logra establecer una matroid que se pueda asociar al problema de los código de Huffman, aunque el algoritmo siempre retorna una solución óptima al problema.

CONCLUSIONES

- La complejidad temporal de un algoritmo voraz generalmente está determinado por la función de selección que se maneje, los algoritmos presentados en este documento en su mayoría utilizan un sistema de ordenamiento ya sea ascendente o descendente que contribuye a la selección de elementos; este ordenamiento (por montículos en este documento) tiene un orden de $n \log n$ y es de resaltar que existen otros tipos de ordenamiento que son más eficientes como el ordenamiento rápido o *Quicksort* el cual puede ser fácilmente implementado.

- Para el problema del árbol de recubrimimento mínimo, la relación entre la complejidad del algoritmo de $Prim$ y la del algoritmo de $Kruskal$ depende fundamentalmente de la cantidad de aristas del grafo; más concretamente, el algoritmo de $Prim$ es más eficiente cuando el grafo tiende a ser completo y el algoritmo de $Kruskal$ cuando el número de aristas del grafo se aproxima a $n-1$, siendo n es el número de nodos del grafo.

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

- La teoría de matroid presenta un marco formal alternativo para la prueba del correcto funcionamiento del algoritmo $Kruskal$ en la solución del problema del árbol de recubrimiento.
- No es posible aplicar la teoría de matroid a los problemas de la mochila, del cambio de monedas y de los recorridos del caballo de ajedrez.
- Por la forma en que se construyó el software, es interactivo, dinámico, seguro, didáctico, ilustrativo y de fácil manejo. De esta manera se espera facilite la comprensión del funcionamiento de los algoritmos voraces en la solución de algunos problemas.

Bibliografía

[1] CORMEN, Thomas; LEISERSON, Charles y RIVEST, Ronald. *Introduction to Algorithms*. Nueva York. McGraw Hill.1990.

[2] MARTIN, James; ODELL James J. *Análisis y diseño orientado a objetos*, México. Prentice Hall Hispanoamericana, 1994.

[3] G. AUSIELLO, P. Crescenzi. *Complexity and approximation, combinatorian optimization problems and their approximability properties*. Springer. 1.999.

[4] G. BRASSARD, P Bratley. *Fundamentos de Algoritmia*. Madrid. Prentice Hall. 1999.

[5] *Algoritmia Básica*, disponible en Internet. http://www.iptl.edu.mx

[6] *Algoritmos Voraces*, disponible en Internet. http://www.iti.upv.es/ evidal/students/ad3/tema4

[7] *Algoritmos Voraces*, disponible en Internet. http://www.isi.upc.edu/ iea/transpasjavier/voracesjavier.ppt.

[8] *Programación Orientada a Objetos*, disponible en Internet. http://lenguajes-de-programacion.com/programacion-orientada-a-objetos.shtml