Capítulo 4

ALGORITMOS ÁVIDOS

4.1 INTRODUCCIÓN

El método que produce algoritmos ávidos es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización.

Dado un problema con *n* entradas el método consiste en obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema. Cada uno de los subconjuntos que cumplan las restricciones diremos que son soluciones *prometedoras*. Una solución prometedora que maximice o minimice una función objetivo la denominaremos solución óptima.

Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo ávido vamos a definir una serie de elementos que han de estar presentes en el problema:

- Un conjunto de *candidatos*, que corresponden a las *n* entradas del problema.
- Una *función de selección* que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una función que compruebe si un cierto subconjunto de candidatos es *prometedor*. Entendemos por prometedor que sea posible seguir añadiendo candidatos y encontrar una solución.
- Una *función objetivo* que determine el valor de la solución hallada. Es la función que queremos maximizar o minimizar.
- Una función que compruebe si un subconjunto de estas entradas es solución al problema, sea óptima o no.

Con estos elementos, podemos resumir el funcionamiento de los algoritmos ávidos en los siguientes puntos:

1. Para resolver el problema, un algoritmo ávido tratará de encontrar un subconjunto de candidatos tales que, cumpliendo las restricciones del problema, constituya la solución óptima.
2. Para ello trabajará por etapas, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá

de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.

3. Antes de añadir un candidato a la solución que está construyendo comprobará si es prometedora al añadilo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato comprobará si el conjunto obtenido es solución.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un esquema general para este tipo de algoritmos:

```
PROCEDURE AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;
   VAR x:ELEMENTO; solucion:CONJUNTO; encontrada:BOOLEAN;
BEGIN
   encontrada:=FALSE; crear(solucion);
   WHILE NOT EsVacio(entrada) AND (NOT encontrada) DO
      x:=SeleccionarCandidato(entrada);
      IF EsPrometedor(x,solucion) THEN
         Incluir(x,solucion);
         IF EsSolucion(solucion) THEN
            encontrada:=TRUE
         END;
      END
   END;
   RETURN solucion;
END AlgoritmoAvido;
```

De este esquema se desprende que los algoritmos ávidos son muy fáciles de implementar y producen soluciones muy eficientes. Entonces cabe preguntarse ¿por qué no utilizarlos siempre? En primer lugar, porque no todos los problemas admiten esta estrategia de solución. De hecho, la búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global, como mostraremos en varios ejemplos de este capítulo. La estrategia de los algoritmos ávidos consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, para ganar la guerra muchas veces es necesario perder alguna batalla.

Desgraciadamente, y como en la vida misma, pocos hechos hay para los que podamos afirmar sin miedo a equivocarnos que lo que parece bueno para hoy siempre es bueno para el futuro. Y aquí radica la dificultad de estos algoritmos. Encontrar la función de selección que nos garantice que el candidato escogido o rechazado en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión. Por ello, una parte muy importante de este tipo de algoritmos es la demostración formal de que la función de selección escogida consigue encontrar óptimos globales para cualquier entrada del algoritmo. No basta con diseñar un procedimiento ávido, que seguro

que será rápido y eficiente (en tiempo y en recursos), sino que hay que demostrar que siempre consigue encontrar la solución óptima del problema.

Debido a su eficiencia, este tipo de algoritmos es muchas veces utilizado aun en los casos donde se sabe que no necesariamente encuentran la solución óptima. En algunas ocasiones la situación nos obliga a encontrar pronto una solución razonablemente buena, aunque no sea la óptima, puesto que si la solución óptima se consigue demasiado tarde, ya no vale para nada (piénsese en el localizador de un avión de combate, o en los procesos de toma de decisiones de una central nuclear). También hay otras circunstancias, como veremos en el capítulo dedicado a los algoritmos que siguen la técnica de Ramificación y Poda, en donde lo que interesa es conseguir cuanto antes una solución del problema y, a partir de la información suministrada por ella, conseguir la óptima más rápidamente. Es decir, la eficiencia de este tipo de algoritmos hace que se utilicen aunque no consigan resolver el problema de optimización planteado, sino que sólo den una solución "aproximada".

El nombre de algoritmos ávidos, también conocidos como voraces (su nombre original proviene del término inglés *greedy*) se debe a su comportamiento: en cada etapa "toman lo que pueden" sin analizar consecuencias, es decir, son glotones por naturaleza. En lo que sigue veremos un conjunto de problemas que muestran cómo diseñar algoritmos ávidos y cuál es su comportamiento. En este tipo de algoritmos el proceso no acaba cuando disponemos de la implementación del procedimiento que lo lleva a cabo. Lo importante es la demostración de que el algoritmo encuentra la solución óptima en todos los casos, o bien la presentación de un contraejemplo que muestra los casos en donde falla.

4.2 EL PROBLEMA DEL CAMBIO

Suponiendo que el sistema monetario de un país está formado por monedas de valores $v_1,v_2,...,v_n$, 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.

En primer lugar, es fácil implementar un algoritmo ávido para resolver este problema, que es el que sigue el proceso que usualmente utilizamos en nuestra vida diaria. Sin embargo, tal algoritmo va a depender del sistema monetario utilizado y por ello vamos a plantearnos dos situaciones para las cuales deseamos conocer si el algoritmo ávido encuentra siempre la solución óptima:

a) Suponiendo que cada moneda del sistema monetario del país vale al menos el doble que la moneda de valor inferior, que existe una moneda de valor unitario, y que disponemos de un número ilimitado de monedas de cada valor.
b) Suponiendo que el sistema monetario está compuesto por monedas de valores 1, p, p^2, p^3,..., p^n, donde $p > 1$ y $n > 0$, y que también disponemos de un número ilimitado de monedas de cada valor.

Solución (𝒮)

Comenzaremos con la implementación de un algoritmo ávido que resuelve el problema del cambio de dinero:

```
TYPE  MONEDAS =(M500,M200,M100,M50,M25,M5,M1);(*sistema monetario*)
      VALORES = ARRAY MONEDAS OF CARDINAL; (* valores de monedas *)
      SOLUCION = ARRAY MONEDAS OF CARDINAL;

PROCEDURE Cambio(n:CARDINAL;VAR valor:VALORES;VAR cambio:SOLUCION);
(* n es la cantidad a descomponer, y el vector "valor" contiene los
valores de cada una de las monedas del sistema monetario *)
   VAR moneda:MONEDAS;
BEGIN
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      cambio[moneda]:=0
   END;
   FOR moneda:=FIRST(MONEDAS) TO LAST(MONEDAS) DO
      WHILE valor[moneda]<=n DO
         INC(cambio[moneda]);
         DEC(n,valor[moneda])
      END
   END
END Cambio;
```

Este algoritmo es de complejidad lineal respecto al número de monedas del país, y por tanto muy eficiente.

Respecto a las dos cuestiones planteadas, comenzaremos por la primera. Supongamos que nuestro sistema monetario esta compuesto por las siguientes monedas:

```
TYPE MONEDAS = (M11,M5,M1);  valor:={11,5,1};
```

Tal sistema verifica las condiciones del enunciado pues disponemos de moneda de valor unitario, y cada una de ellas vale más del doble de la moneda inmediatamente inferior.

Consideremos la cantidad $n = 15$. El algoritmo ávido del cambio de monedas descompone tal cantidad en:

$$15 = 11 + 1 + 1 + 1 + 1,$$

es decir, mediante el uso de cinco monedas. Sin embargo, existe una descomposición que utiliza menos monedas (exactamente tres):

$$15 = 5 + 5 + 5.$$

Aunque queda comprobado que bajo estas circunstancias el diseño ávido no puede utilizarse, las razones por las que el algoritmo falla quedarán al descubierto cuando analicemos el siguiente punto.

b) En cuanto a la segunda situación, y para demostrar que el algoritmo ávido encuentra la solución óptima, vamos a apoyarnos en una propiedad general de los números naturales:

Si p es un número natural mayor que 1, todo número natural x puede expresarse de forma única como:

$$x = r_0 + r_1 p + r_2 p^2 + ... + r_n p^n, \qquad [4.1]$$

con $0 \le r_i < p$ para todo $0 \le i \le n$ y siendo n el menor natural tal que $x < p^{n+1}$, es decir, $n = \lfloor \log_p x \rfloor$.

El algoritmo del cambio de monedas lo que hace en nuestro caso es calcular los r_i, que indican el número de monedas a devolver de valor p^i ($0 \le i \le n$). Lo que tenemos que demostrar es que esa descomposición es óptima, esto es, que si

$$x = s_0 + s_1 p + s_2 p^2 + ... + s_m p^m$$

es otra descomposición distinta, entonces:

$$\sum_{i=0}^{n} r_i < \sum_{i=0}^{m} s_i \ .$$

Para realizar esta demostración lo haremos primero para $p = 2$ porque intuitivamente resulta más sencillo de entender el proceso de la demostración. El caso general resulta ser análogo.

Sea entonces

$$x = r_0 + 2r_1 + 2^2 r_2 + ... + 2^n r_n \qquad [4.2]$$

la descomposición obtenida por el algoritmo ávido. Por tanto $x < 2^{n+1}$ y los coeficientes r_i toman los valores 0 ó 1. Consideremos además otra descomposición distinta:

$$x = s_0 + 2s_1 + 2^2 s_2 + ... + 2^m s_m.$$

Paso 1:

En primer lugar, como se verifica que $x < 2^{n+1}$, esto implica que $m \le n$. Definimos entonces $s_{m+1} = s_{m+2} = ... = s_n = 0$ para poder disponer de n términos en cada descomposición.

Paso 2:

Queremos ver que

$$r_0 + r_1 + ... + r_n < s_0 + s_1 + ... + s_n.$$

Como ambas descomposiciones son distintas, sea k el primer índice tal que $r_k \ne s_k$. Podemos suponer sin perder generalidad que $k = 0$, puesto que si no lo fuera podríamos restar a ambos lados de la desigualdad los términos iguales y dividir por la potencia de 2 adecuada. Veamos que si $r_0 \ne s_0$ entonces $r_0 < s_0$.

- Si x es par entonces $r_0 = 0$. Como $s_0 \ge 0$ y estamos suponiendo que $r_0 \ne s_0$, s_0 ha de ser mayor que cero y por tanto podemos deducir que $r_0 < s_0$.

- Si x es impar entonces $r_0 = 1$. Pero en la segunda descomposición de x también ha de haber al menos una moneda de una unidad, y por tanto $s_0 \geq 1$. Al estar suponiendo que $r_0 \neq s_0$, podemos deducir también aquí que $r_0 < s_0$.

Con esto, consideremos la cantidad $s_0 - r_0 > 0$. Tal cantidad ha de ser par pues $x - r_0$ lo es (por la expresión [4.2]). Y por ser par, siempre podremos "mejorar" la segunda descomposición $(s_0,s_1,...,s_n)$ cambiando $s_0 - r_0$ monedas de 1 unidad por $(s_0 - r_0)/2$ monedas de 2 unidades, obteniendo:

$$s_0 + s_1 + \ldots + s_n > r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) + s_2 + \ldots + s_n . \qquad [4.3]$$

Paso 3:

Mediante el razonamiento anterior hemos obtenido una nueva descomposición, mejor que la segunda, y manteniendo además que:

$$r_0 + \left(s_1 + \frac{s_0 - r_0}{2} \right) 2 + s_2 2^2 + \ldots + s_n 2^n = x = r_0 + r_1 2 + \ldots + r_n 2^n .$$

Podemos volver a aplicar el razonamiento del paso 2 sobre esta nueva descomposición, y así sucesivamente ir viendo que $s_i \geq r_i$ para todo $0 \leq i \leq n-1$, e ir obteniendo nuevas descomposiciones, cada una mejor que la anterior, hasta llegar en el último paso a una descomposición de la forma:

$$r_0 + r_1 2 + r_2 2^2 + \ldots + r_{n-1} 2^{n-1} + \left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) 2^n = x \qquad [4.4]$$

en la que hemos ido acumulando las diferencias en el último término, y que además verifica que:

$$r_0 + r_1 + \ldots + r_i + \left(s_{i+1} + \frac{s_i - acum_i}{2} \right) + \ldots + s_n \geq r_0 + r_1 + \ldots + r_n , \; (0 \leq i \leq n-1)$$

Una vez llegado a este punto la demostración está ya realizada, puesto que si se verifica [4.4], por la unicidad de la descomposición a la que hacía referencia la propiedad [4.1], se ha de cumplir que

$$\left(s_n + \frac{s_{n-1} - acum_{n-1}}{2} \right) = r_n ,$$

y esto, junto a la cadena de desigualdades [4.3], hace que sea cierta nuestra afirmación. Para el caso $p > 2$ el razonamiento es igual.

Resta sólo preguntarnos por qué esta demostración no funciona para cualquier sistema monetario. La razón fundamental se encuentra en la expresión [4.4], que en este sistema permite pasar monedas de una unidad a otra sin problemas, no siendo válido para todos.