Algorítmica

Tema 1. La Eficiencia de los Algoritmos

Tema 2. Algoritmos “Divide y Vencerás”

Tema 3. Algoritmos Voraces (“Greedy”)

Tema 4. Algoritmos para la Exploración de Grafos (“Backtraking”, “Branch and Bound”)

Tema 5. Algoritmos basados en Programación Dinámica

Tema 6. Otras Técnicas Algorítmicas de Resolución de Problemas

Objetivos

- Comprender la filosofía de diseño de los algoritmos voraces
- Conocer las características de un problema resoluble mediante un algoritmo voraz
- Resolución de diversos problemas
- Heurísticas voraces: Soluciones aproximadas a problemas

Índice

- **EL ENFOQUE GREEDY**
- **ALGORITMOS GREEDY EN GRAFOS**
- **HEURÍSTICA GREEDY**

Índice

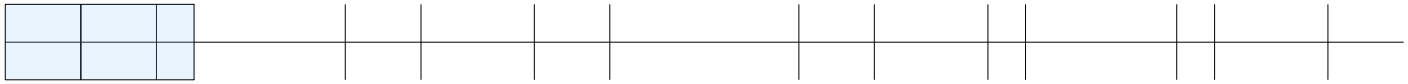
La filosofía Greedy (voraz)

- Buscan siempre la mejor opción en cada momento
- La decisión se toma en base a criterios locales

El termino **greedy**
es sinónimo de voraz, ávido, glotón, ...

Selección de puntos de Parada

- Un camión va desde Granada a Moscú siguiendo una ruta predeterminada. Se asume que conocemos las gasolineras que se pueden encontrar en la ruta.
- La capacidad del depósito es = C. 



- Problema: Minimizar el número de paradas que hace el conductor

¿Cómo se aplicaría la idea anterior para resolver este problema?

Algoritmos Greedy (voraz)

¡Comete siempre todo lo que tengas a mano!

En este problema

Avanza lo más que puedas antes de rellenar el depósito.

Selección de puntos de parada

```
Ordenar las gasolineras en orden creciente
0 = g0 < g1 < g2 < ... < gn.
set<gasolineras> S;  // Seleccionamos gasolineras
x = g0
while (x != gn)
  gp = mayor gasolinera t.q. gp <= (x + C)
  if (gp = x) return "no hay solución"
  else {x = gp
        S.push_back(gp)
  }
return S, S.size();
```

Características generales de los algoritmos voraces

- Se utilizan generalmente para resolver problemas de optimización: máximo o mínimo.
- Un algoritmo “greedy” toma las decisiones en función de la información local que está disponible en cada momento (“miopes”).
- Una vez tomada la decisión no se la vuelve a replantear en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No garantizan alcanzar la solución óptima.

Elementos de un algoritmo voraz

Conjunto de Candidatos (C) : representa al conjunto de posibles decisiones que se pueden tomar en cada momento.

Conjunto de Seleccionados (S): representa al conjunto de decisiones tomadas hasta este momento.

Función Solución: determina si se ha alcanzado una solución (no necesariamente óptima).

Función de Factibilidad: determina si es posible completar el conjunto de candidatos seleccionados para alcanzar una solución al problema (no necesariamente óptima).

Función Selección: determina el candidato más prometedor del conjunto a seleccionar.

Función Objetivo: da el valor de la solución alcanzada.

Selección de puntos de parada

```
Ordenar las gasolineras en orden creciente

0 = g0 < g1 < g2 < ... < gn.

set<gasolineras> S;   // Seleccionamos gasolineras

x = g0

while (x != gn)

  gp = mayor gasolinera t.q. gp <= (x + C)

  if (gp = x) return "no hay solución"

  else { x = gp

         S.push_back(gp)

  }

return S, S.size()
```

Candidatos

Seleccionados

F. Solución

F. Selección

F. Factibilidad

F. Objetivo

Esquema de un algoritmo voraz

Un algoritmo Greedy procede siempre de la siguiente manera:

- Se parte de un conjunto de candidatos a solución vacío: S = ∅
- De la lista de candidatos que hemos podido identificar, con la función de selección, se coge el mejor candidato posible
- Vemos si con ese elemento podríamos llegar a constituir una solución: Si se verifican las condiciones de factibilidad en S
- Si el candidato anterior no es válido, lo borramos de la lista de candidatos posibles, y nunca más es considerado
- Evaluamos la función objetivo. Si no estamos en el óptimo seleccionamos con la función de selección otro candidato y repetimos el proceso anterior hasta alcanzar la solución.

La primera solución que se consigue suele ser la solución óptima del problema.

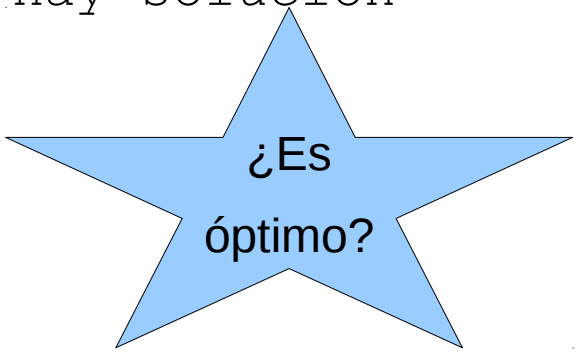
Esquema de un algoritmo voraz

```
Voraz(C : conjunto de candidatos) : conjunto solución
    S = Ø
    mientras C ≠ Ø y no Solución(S) hacer
        x = Seleccion(C)
        C = C – {x}
        si factible(S U {x}) entonces
            S = S U {x}
        fin si
    fin mientras
    si Solución(S) entonces
        Devolver S
    en otro caso
        Devolver "No se encontró una solución"
    fin si
```

El enfoque Greedy suele proporcionar soluciones óptimas, pero no hay garantía de ello. Por tanto, siempre habrá que estudiar la **corrección del algoritmo** para verificar esas soluciones

Esquema de un algoritmo voraz

```
Ordenar las gasolineras en orden creciente
0 = g0 < g1 < g2 < ... < gn.
set<gasolineras> S;  // Seleccionamos gasolineras
x = g0
while (x != gn)
  gp = mayor gasolinera t.q. gp <= (x + C)
  if (gp = x) return "no hay solución"
  x = gp
  S.push_back(gp)
return S
```



Demostraciones

- ¿Por qué? Debemos garantizar que el algoritmo alcanza la solución óptima.
- ¿Cómo?, entre otras utilizaremos la reducción al absurdo (y la inducción)

Para probar que una proposición es verdadera, se supone que es falsa y se llega a un absurdo o a una contradicción, concluyéndose entonces que la proposición debe ser verdadera, pues no puede ser falsa.

Demostrar que una proposición es verdadera demostrando que no puede ser falsa.

Ejemplo Sudoku

1	2	3						
4	5	6						
7	X							
		9						

Proposición:

S[3,2]!=8

Demostración:

Asumir que S[3,2] = 8 y razonar

Hasta alcanzar un absurdo.

Si S[3,2] = 8, entonces, por

Regla del Cuadrante S[3,3]=9

Contradicción con que S[3,3]!=9 por regla Columna

Optimalidad Selec. paradas

- Demostración (red. absurdo):

Sean 0 = g0 < g1 < . . . < gp = L las gasol. seleccionadas por el alg. Greedy. Asumamos que L no es óptimo

Sobre todas las soluciones óptimas 0 = f0 < f1 < . . . < fq (q < p), llamemos r al máximo valor posible donde f0 = g0, f1= g1 , . . . , fr = gr. Sea Lop una de estas soluciones

Entonces, tenemos que

1) g(r+1) > f(r+1) (por como el algoritmo greedy selecciona las gasol. g).

2) 0 = g0 < . . < gr < g(r+1) < f(r+2) < ...< fq es otra solución al problema.

3) Además es óptima (tiene el mismo tamaño que Lop).

Luego Alcanzamos una contradicción: r NO es el máximo valor posible donde se alcanza la igualdad entre L y Lop

Problema de dar cambio

Se desea dar cambio usando el menor número posible de monedas de 1, 5, 10 y 25

¿Es greedy el problema?:

Candidatos: 1, 5, 10, 25 (con una moneda de cada tipo por lo menos).

Seleccionados: Podremos definirlo.

Solución: Lista de candidatos tal que la suma de los mismos coincida exactamente con el cambio pedido.

Criterio de factibilidad: Que no se supere el cambio.

Criterio de selección: Se escoge la moneda de mayor valor entre las disponibles.

Objetivo: El número de monedas ha de ser mínimo.

Problema de dar cambio

Si tenemos, por ejemplo, una moneda de 100 que queremos cambiar y la máquina dispone de 3 monedas de 25, 1 de 10, 2 de 5 y 25 de 1, entonces la primera solución que alcanza el algoritmo greedy directo es justamente la Solución óptima: 3 de 25, 1 de 10, 2 de 5 y 5 de 1.

Pero si tenemos 10 monedas de 1, 5 de 5, 3 de 10, 3 de 12 y 2 de 25, la solución del algoritmo para una moneda de 100 sería 2 de 25, 3 de 12, 1 de 10 y 4 de 1, en total diez monedas.

La solución no es óptima, ya que existe otra mejor que utiliza nueve monedas en vez de diez, que es 2 de 25, 3 de 10 y 4 de 5.

Para demostrar la no optimalidad basta un contraejemplo.

Problema Selección Programas

- Dado un conjunto T de n programas, cada uno con tamaño t1, ..., tn y un dispositivo de capacidad máxima C

- Objetivo:
  - Seleccionar el mayor número de programas que se pueden almacenar en C.

Problema: SelPro(T,C)

Problema Selección Programas: Solución Greedy

- *Candidatos a seleccionar:*
  - Programas
- *Candidatos seleccionados*
- *Función Solución*:
  - No entran más candidatos en el dispositivo
- *Función de Factibilidad*:
  - Es posible incluir el candidato actual en el dispositivo
- *Función Selección*: determina el mejor candidato del conjunto a seleccionar.
  - Seleccionar el candidato de menor tamaño
- *Función Objetivo*:
  - Número de programas en el dispositivo

Demostración técnica greedy alcanza el óptimo:

Tenemos que demostrar que

Optimo Local $\Rightarrow$ Solución global optimal

- Paso 1.
  - Demostrar que la primera decisión es correcta.
- Paso 2.
  - Demostrar que existen subestructuras optimales.
    - Es decir, cuando se ha tomado la decisión #1, el problema se reduce a encontrar una solución optimal para el sub-problema que tiene como candidatos los compatibles con #1

Dem.: Primera decisión es correcta

- Suponemos los programas ordenados en tamaño

$$t1 <= t2 <= t3 <= \ldots <= tn$$

- Teorema: Si T es un conjunto de programas (ordenado), entonces $\exists$ una solución optimal $A \subseteq T$ tal que $\{1\} \in A$
  - Idea: Usar reducción al absurdo
  - Si ninguna solución optimal contiene {1}, elegimos una, B, y siempre podremos reemplazar la primera actividad en B por {1} (xq?). Obteniendo el mismo número de programas, y por tanto una sol. optimal.

Dem.: subestructuras optimales

- Teorema: Sea *A* una solución óptima al problema *SelPro(T,C) y sea t1 el primer programa en A* . Entonces *A* - {*t1*} es solución óptima para SelPro(T*,C-t1), con T* = {t*i en T*: i>1}
  - Demostración: (Red. Absurdo)
  - Partimos de que *A* – {*t1*} NO ES solución óptima para SelPro(T*,C-t1) . Esto es, podemos encontrar una solución optimal *B* al problema SelPro(T*,C-t1) donde se verifica que |*B*| > |*A* - {*t1*}|,
    - Entonces:
    - ?? *B* U {*t1*} es solución a SelPro(T,C)
    - Pero |*B* U {*t1*}| > |A| !!! Contradicción con el hecho de que A es solución optimal al problema SelPro(T,C)

Problema Selección de Actividades

Tenemos la entrada de una Exposición que organiza un conjunto de actividades

- Para cada actividad conocemos su horario de comienzo y fin.
- Con la entrada podemos asistir a todas las actividades.
- Hay actividades que se solapan en el tiempo.

Objetivo: Asistir al mayor número de actividades posible =>
Problema de selección de actividades

Otra alternativa: Minimizar el tiempo que estamos ociosos.

Problema Selección de Actividades

Dado un conjunto *S* de *n* actividades

si = tiempo de comienzo de la actividad *i*

fi = tiempo de finalización de la actividad *i*

– Encontrar el subconjunto de actividades compatibles A de tamaño máximo

Problema Selección de Actividades

Fijemos los elementos de la técnica

- *Candidatos a seleccionar: Conj. Actividades, S*
- *Candidatos seleccionados: Conjunto A, inic. A={Ø }*
- *Función Solución: S={Ø }.*
- *Función Selección: determina el mejor candidato, x*
  - *– Menor duración.*
  - *– Menor solapamiento.*
  - *– Comienza antes.*
  - *– Termina antes.*
- *Función de Factibilidad: x es factible si es compatible con las actividades en A (no hay actividades solapadas).*
- *Función Objetivo: Tamaño de A.*

Selección de actividades: contraejemplos

- Comienza antes

- Menor duración

- Menor solapamiento

Problema Selección de Actividades

Algoritmo Greedy

- *SelecciónActividades(Activ S,A)*
  - *Ordenar S en orden creciente de tiempo de finalización*
  - *Seleccionar la primera actividad.*
  - *Repetir*

    Seleccionar la siguiente actividad en el orden que comience después de que la actividad previa termine.
  - *Hasta que S esté vacio.*

Selección de actividades

```
SelecciónActividadesGreedy(S, A){
qsort(S,n); // según tiempo de finalización
A[0]= S[0] // Seleccionar primera actividad
i=1; prev = 0;

while (!S.empty()) { // es solucion(S)
    x = S[i] // seleccionar;
    if (x.inicio > A[prev].fin) // factible x
        A[prev++] = x; // insertamos en solucion
    else i++; // rechazamos
    }
 }
```

Problema Selección de Actividades

- ***¿Optimalidad?***

T.H. CORMEN, C.E. LEISERSON, R.L. RIVEST.
Introduction to Algorithms. The MIT Press (1992)

Se puede probar de forma similar a la del problema de las gasolineras.

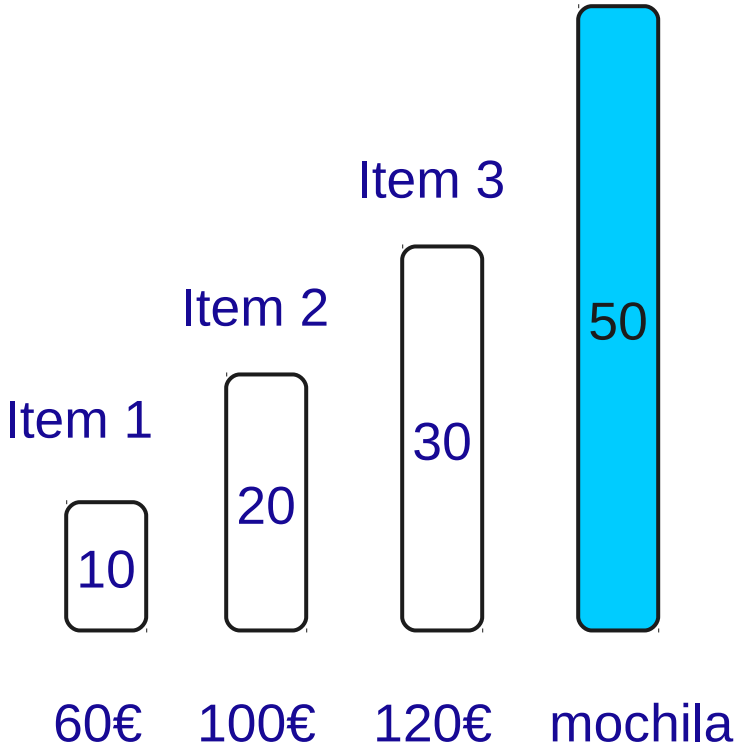
Problema de la mochila fraccional

- Consiste en llenar una mochila:
  - Puede llevar como máximo un peso *P*
  - *Hay n distintos posibles* objetos i fraccionables cuyos pesos son p_i y el beneficio por cada uno de esos objetos es de b_i. Si incluimos una fracción x_i ($0<=x_i<=1$) del objeto i se genera un beneficio de $x_i b_i$ y un peso de $x_i p_i$.

- El *objetivo* es maximizar el beneficio de los objetos transportados.

$$\max \sum_{i=1}^{n} x_i b_i$$
$$s.a. \sum_{i=1}^{n} x_i p_i \leq P$$

Ejemplo



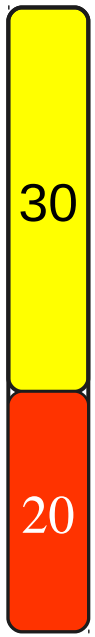
Es un claro problema de tipo greedy

Sus aplicaciones son inumerables

Es un banco de pruebas algorítmico

La técnica greedy produce soluciones optimales para este tipo de problemas cuando se permite fraccionar los objetos

Ejemplo: Mochila 0/1



120€

100€

100€

60 €

120€

60 €

Total = 220 € = 160 € = 180 €

¿Cómo seleccionamos los items?

Ejemplo: Problema de la mochila fraccional

- Supongamos 5 objetos de peso y precios dados por la tabla, la capacidad de la mochila es 100.

Precio (euros)	20	30	65	40	60
Peso (kilos)	10	20	30	40	50

- **Método 1 elegir primero el menos pesado**
  - Peso total = 1*10 + 1*20 + 1*30 + 1*40 = 100
  - Beneficio total = 20 + 30 + 65 + 40 = 155
- **Método 2 elegir primero el de más beneficio**
  - Peso Total = 1*30 + 1*50 + 0.5*40 = 100
  - Beneficio Total = 65 + 60 + 0.5*40 = 145

Ejemplo: Problema de la mochila fraccional

- Metodo 3 elegir primero el que tenga mayor valor por unidad de peso (razón beneficio / peso)

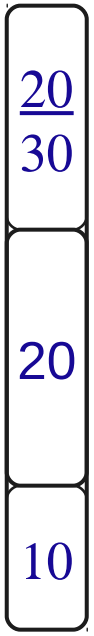
Precio (euros)	20	30	65	40	60
Peso (Kilos)	10	20	30	40	50
Precio/Peso	2	1,5	2,1	1	1,2

  - Peso Total = 1*30 + 1*10 + 1*20 + 0.8*50 = 100
  - Benef. Total = 65 + 20 + 30 + 0.8*60 = 163

Mochila fraccional

Tomando los items en orden de mayor valor por unidad de peso, se obtiene una solución optimal

{60/10, 100/20, 120/30}



80 €

100€

60 €

Total = 240 €

Solución Greedy

- Definimos la densidad del objeto A_i por b_i/p_i.
- Se usan objetos de tan alta densidad como sea posible, es decir, los seleccionaremos en orden decreciente de densidad.
- Si es posible se coge todo lo que se pueda de A_i, pero si no se rellena el espacio disponible de la mochila con una fracción del objeto en curso, hasta completar la capacidad, y se desprecia el resto.
- Se ordenan los objetos por densidad no creciente, i.e.:

$$b_i/p_i \geq b_{i+1}/p_{i+1} \text{ para } 1 \leq i < n.$$

Pseudocódigo mochila

Procedimiento MOCHILA_GREEDY(b,p,M,X,n)

//b(1:n) y p(1:n) contienen los beneficios y pesos respectivos de los n objetos ordenados como b(i)/p(i) $\geq$ b(i+1)/p(i+1). M es la capacidad de la mochila y X(1:n) es el vector solución

X = 0; //inicializa la solucion en cero

cr = M; // cr = capacidad restante de la mochila

i=1;

While (p(i) <= cr and i <= n)

    X(i) = 1;

    cr = cr – p(i);

    i = i+1;

if i<= n Entonces X(i) = cr/p(i)

Demostración de optimalidad

Vamos a demostrar que el algoritmo siempre encuentra la solución optimal del problema

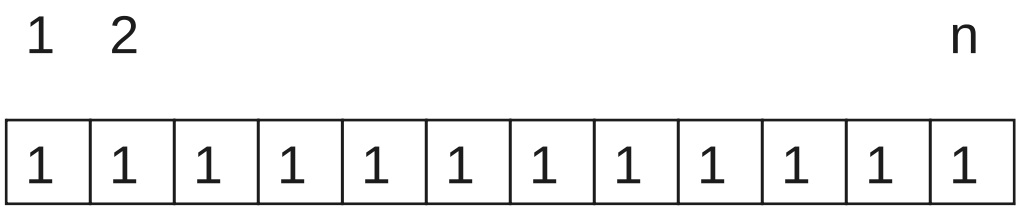
Sea $b_1/p_1 \geq b_2/p_2 \geq \ldots \geq b_n/p_n$

Sea $X = (x_1,x_2,\ldots,x_n)$ la solución generada por MOCHILA_GREEDY, para una capacidad M

Sea $Y = (y_1,y_2,\ldots,y_n)$ una solución factible cualquiera

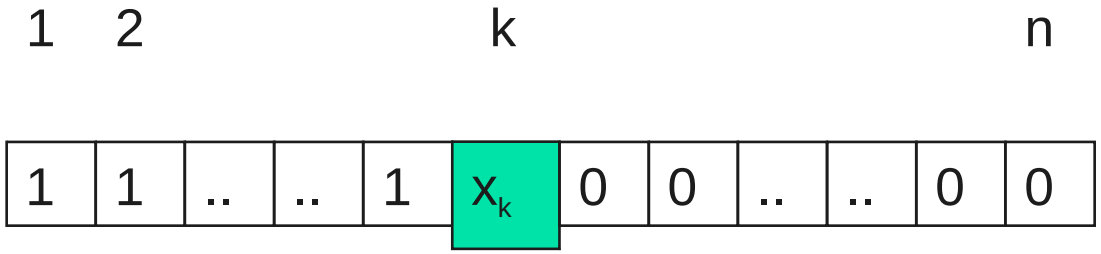
Queremos demostrar que $\sum_{i=1}^{n} (x_i - y_i) b_i \geq 0$

Demostración de optimalidad

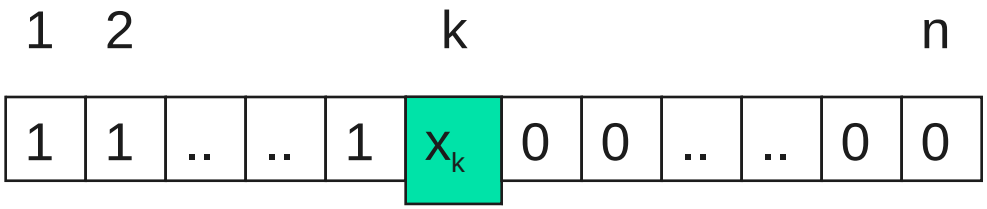


Si todos los x_i son 1, la solución es claramente optimal.

En otro caso, sea k el menor número tal que $x_k < 1$.



Demostración de optimalidad



$$\sum_{i=1}^{n}(x_i-y_i)b_i = \sum_{i=1}^{k-1}(x_i-y_i)\,p_i\frac{b_i}{p_i} + (x_k-y_k)\,p_k\frac{b_k}{p_k}$$

$$+\sum_{i=k+1}^{n}(x_i-y_i)\,p_i\frac{b_i}{p_i}$$

Demostración de optimalidad

- $x_i - y_i >= 0$ y $b_i/p_i >= b_k/p_k$

$$\sum_{i=1}^{k-1} (x_i - y_i) p_i \frac{b_i}{p_i} \geq \sum_{i=1}^{k-1} (x_i - y_i) p_i \frac{b_k}{p_k}$$

$$(x_k - y_k) p_k \frac{b_k}{p_k} = (x_k - y_k) p_k \frac{b_k}{p_k}$$

- $x_i - y_i <= 0$ y $b_i/p_i <= p_k/p_k$

$$\sum_{i=k+1}^{n} (x_i - y_i) p_i \frac{b_i}{p_i} \geq \sum_{i=k+1}^{n} (x_i - y_i) p_i \frac{b_k}{p_k}$$

Demostración de optimalidad

$$\sum_{i=1}^{n}(x_i - y_i)b_i \geq \sum_{i=1}^{n}(x_i - y_i)p_i\frac{b_k}{p_k}$$

$$\frac{b_k}{p_k}\sum_{i=1}^{n}(x_i - y_i)p_i$$

$\sum_i x_i p_i = M$ por hipótesis, pero $\sum_i y_i p_i \leq M$. Luego

$$\sum_{i=1}^{n}(x_i - y_i)b_i \geq 0$$