Diseño de algoritmos

Algoritmos de Vuelta Atrás

Jesús Bermúdez de Andrés

Universidad del País Vasco/Euskal Herriko Unibertsitatea **(UPV/EHU)**

Curso 2008-09

1. Algoritmos de vuelta atrás
   - El coloreado de mapas
   - Número mínimo de monedas
   - Suma de subconjunto
   - El problema de la mochila

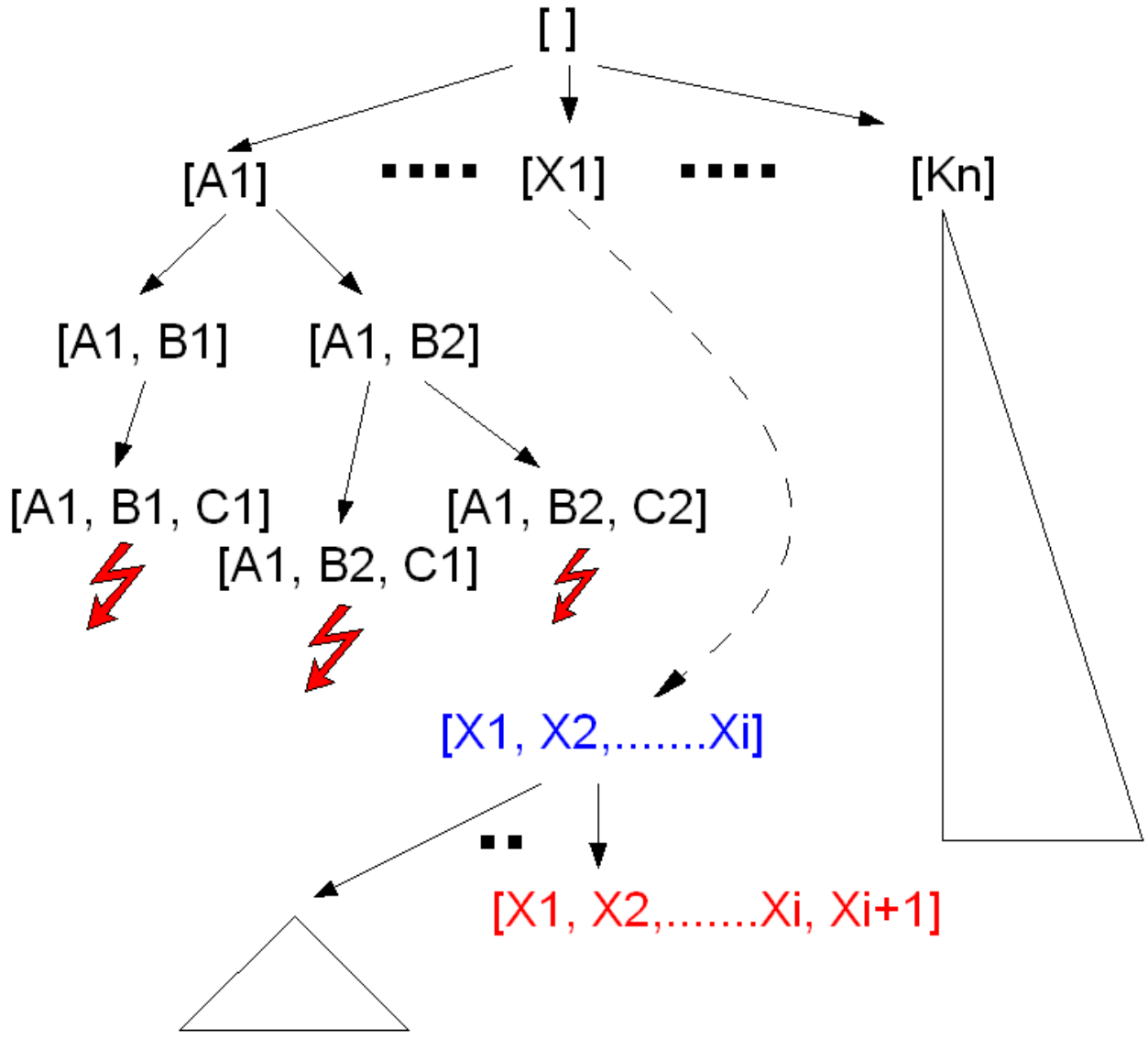
Algoritmos de vuelta atrás (1/2)

Hay problemas para los que no se conoce mejor solución que la que consiste, básicamente, en una búsqueda sistemática en un espacio de propuestas posibles de solución.
La técnica de *vuelta atrás* consiste en construir la solución de un problema mediante extensiones de *ensayos* parciales de solución.

En general, se parte del ensayo neutro [ ] y en un estado intermedio del proceso, tras haber construido el ensayo parcial $[x_1, \ldots, x_i]$, se tantea sistemáticamente con cada posibilidad x_{i+1} de extender el ensayo:

- si $[x_1, \ldots, x_i, x_{i+1}]$ es un ensayo *aceptable*, se extiende $[x_1, \ldots, x_i]$ a $[x_1, \ldots, x_i, x_{i+1}]$ y se continúa con este último ensayo parcial.
- si ninguna de las posibilidades para x_{i+1} hacen que $[x_1, \ldots, x_i, x_{i+1}]$ sea *aceptable* entonces significa que el ensayo $[x_1, \ldots, x_i]$ no es extensible a solución, por consiguiente retiramos la última opción x_i añadida a ese ensayo y continuamos con el ensayo parcial $[x_1, \ldots, x_{i-1}]$ tanteando otra posible opción.

Árbol de ensayos



Algoritmos de vuelta atrás (2/2)

Esta estrategia equivale al recorrido en profundidad de un árbol implícito (ya que ese árbol no está representado por ninguna estructura de datos) comenzando desde la raíz, tal que cada nodo se corresponde con un ensayo parcial y cada arista que parte de un nodo se corresponde con una opción de tanteo.

Cuando un ensayo no es aceptable equivale a podar la posible rama del árbol que pudiera crecer desde ese nodo. Cuanto más "inteligente" sea la poda más eficiente será el algoritmo.

Las hojas de ese árbol implícito se corresponden con ensayos no extensibles a solución o bien con ensayos que son solución.

Esquema de vuelta atrás que calcula todas las soluciones

proc ENSAYAR (E)
  if SOLUCIÓN(E) **then**
    "gestionar el E solución"
  {**else** }
    for cada posibilidad p de extender E **loop**
      if ES_ACEPTABLE$(E + p)$ **then**
        ENSAYAR $(E + p)$

Debe retirarse la cláusula {**else** } en caso de que haya soluciones que sean prefijos de otras soluciones.

Esquema de vuelta atrás que calcula la primera solución

```
proc Ensayar (E, éxito)
  if Solución(E) then
    "gestionar el E solución"
    éxito ← true
  else
    for cada posibilidad p de extender E loop
      if Es_aceptable(E + p) then
        Ensayar (E + p, éxito)
        if éxito then return
```

Análisis de un algoritmo de vuelta atrás

Un modo de aproximarse al análisis de la eficiencia de los algoritmos de vuelta atrás es analizar el árbol implícito de ensayos explorado por el algoritmo:

1. contar el número de ensayos explorados.
2. analizar el coste de calcular las posibilidades de extender cada ensayo y el coste de generar cada nuevo ensayo.
3. analizar el coste de comprobación de la aceptación de un ensayo.

Hay que observar que el resultado es una cota superior.

El rendimiento real del algoritmo puede ser mucho mejor, ya que la función de aceptación de ensayos puede podar una cantidad considerable de ramas del árbol implícito. De hecho, la utilidad de los algoritmos de vuelta atrás depende principalmente de la capacidad de aplicación de funciones de aceptación que poden convenientemente ramas del árbol de ensayos.

Evaluación usando un Método de Montecarlo.

El coloreado de mapas

Calcular todas las formas posibles de colorear los países de un mapa político, usando sólo cuatro colores, de manera que ningún par de países fronterizos esté pintado del mismo color.

Representamos el mapa mediante un grafo no dirigido en el que los nodos representan a los países y cada arista representa que sus extremos son países fronterizos. En particular, representaremos un mapa con n países mediante una matriz de adyacencia $F(1..n, 1..n)$ con valores booleanos.

Árbol de ensayos para CUATRO_COLORES

Un ensayo de longitud k consiste en la lista de colores (tomados de los cuatro posibles) que se han asignado a los países numerados con $[1, \ldots, k]$.

Representamos los ensayos con un vector $color(1..n)$ con valores en el conjunto $\{1, 2, 3, 4\}$ que representa a los cuatro colores. En cada momento del proceso sólo son representativos los k primeros componentes del vector *color*.

Cuando k llegue a ser n y se satisfaga la condición de aceptación tendremos una solución del problema.
Una extensión de un ensayo de longitud k consiste en dar color al país número $k + 1$. Supongamos que tenemos un ensayo de longitud k que satisface las restricciones del problema, entonces una extensión de ese ensayo es aceptable si el color asignado al país número $k + 1$ es distinto del color asignado a sus adyacentes que estén numerados del 1 al k.

Algoritmo Cuatro_colores (1/2)

La llamada inicial debe ser Cuatro_colores(0, F, *color*)

```
proc Cuatro_colores(k, F(1..n, 1..n), color(1..n))
{k: número de países coloreados}
  if k = n then
    Imprimir(color(1..n))
  else
    for C in 1..,4 loop
    color(k + 1) ← C
    if Es_aceptable(F, color, k + 1) then
      Cuatro_colores(k + 1, F, color)
```

Algoritmo Cuatro_colores (2/2)

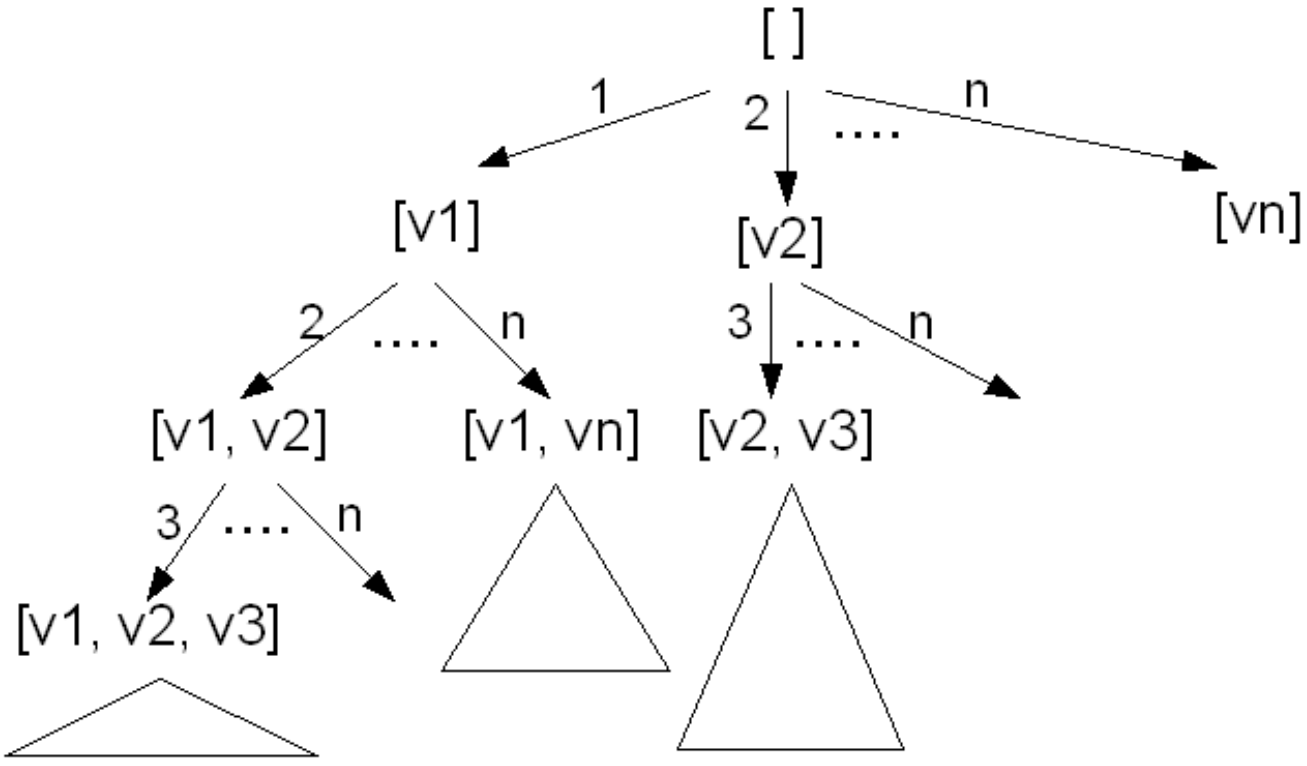
```
func Es_aceptable(F, color, último) return boolean
{último: índice del último país coloreado}
   acept ← true
   país ← 1
   while país < último ∧ acept loop
      if F(país, último) ∧ color(país)=color(último) then
         acept ← false
      end if
      país ← país + 1
   end loop
   return acept
```

Número mínimo de monedas (1/2)

Dadas n de monedas, de valores $v_1, \ldots, v_n$ respectivamente, determinar cual es el número mínimo de monedas que necesitamos para sumar una determinada cantidad C.

- Un ensayo será una secuencia de los valores de esas monedas, que denominaremos *SEC*. Para completar la representación de un ensayo tendremos: una variable *suma* que representa la suma de los valores de *SEC*, una variable *long* que representa la longitud de *SEC* y una variable k que representa el subíndice de la moneda que hay que considerar a continuación.
- La parte del esquema de vuelta atrás dedicada a *"gestionar el ensayo solución"* sirve para memorizar la mejor solución encontrada hasta el momento.
- La solución óptima será la que quede registrada en la secuencia *OPT*, de longitud *longOPT*.

Árbol de ensayos para Número mínimo de monedas



Número mínimo de monedas (2/2)

Suponemos ordenados los valores de monedas, de manera que

$$v_1 \leq v_2 \leq \ldots \leq v_n$$

- Ensayo: Un secuencia de valores de monedas
- Las posibilidades de extender un ensayo $[v_i, \ldots, v_{k-1}]$ son $v_k \ldots v_n$
- El ensayo $[v_i, \ldots, v_{k-1}, v_k]$ es aceptable si su longitud es menor que $longOPT$ y su suma menor o igual que C
- Un ensayo es solución si su suma es igual a C

Algoritmo MinMonedas

Para podar algunas ramas del árbol de ensayos, asumimos que tenemos ordenadas las monedas de manera que $v_1 \leq \ldots \leq v_n$.

proc MinMonedas(SEC, k, $long$, $suma$, OPT, $longOPT$)
    if $suma = C$ **then**
        if $long < longOPT$ **then**
            $OPT \leftarrow SEC$
            $longOPT \leftarrow long$
    else
        while $(k \leq n) \wedge (1 + long < longOPT) \wedge (suma + v_k \leq C)$ **loop**
            $SEC[1 + long] \leftarrow v_k$
            MinMonedas(SEC, $k + 1$, $1 + long$, $suma + v_k$, OPT, $longOPT$)
            $k \leftarrow k + 1$

La llamada inicial debe ser MinMonedas(SEC, 1, 0, 0, OPT, $n + 1$)

Variación de Número mínimo de monedas

¿Qué modificaciones debemos hacer en el algoritmo MinMonedas si disponemos de tantas monedas como se desee de cada valor $v_1, \ldots, v_n$?

Una vez analizada la modificación del árbol de ensayos, observamos que basta con sustituir el valor de $k+1$ por k en la llamada recursiva, es decir: MinMonedas(SEC, k, $1 + long$, $suma + v_k$, OPT, $longOPT$)

Suma de subconjunto

Disponemos de valores $V = \{V_1, \ldots, V_n\}$ enteros positivos y queremos saber todas las formas posibles de sumar D con subconjuntos de V.

- Un ensayo constará de un vector *conjunto* de ceros y unos, de manera que $conjunto[i] = 1$ representa que V_i pertenece al subconjunto seleccionado y $conjunto[i] = 0$ representa que no.
- Tendremos dos variables: *suma* que representa la suma de los valores del subconjunto seleccionado y *resto* que representa la suma de los valores que quedan todavía por considerar.
- La variable j representa el subíndice del valor de V que hay que considerar a continuación.
- Para podar más ramas es importante considerar los valores de V en orden creciente: Suponemos que $V_1 \leq \ldots \leq V_n$.
- Un ensayo es aceptable si $suma + resto \geq D$ (para que sea posible llegar a sumar D) y $suma \leq D$ (para que la suma del ensayo no exceda de la cantidad a sumar).

Algoritmo SumaSubconjunto

proc SumaSubconjunto($suma$, $resto$, j, $conjunto$)
  if $suma = D$ **then**
    Imprimir($conjunto[1..j-1]$)
  else
    if $suma + V_j \leq D$ **then**
      $conjunto[j] \leftarrow 1$
      SumaSubconjunto($suma + V_j$, $resto - V_j$, $j+1$, $conjunto$)
      if $suma + resto - V_j \geq D$ **then**
        $conjunto[j] \leftarrow 0$
        SumaSubconjunto($suma$, $resto - V_j$, $j+1$, $conjunto$)

La llamada inicial debe ser
SumaSubconjunto(0, $\sum_{i=1}^{n} V_i$, 1, $conjunto$)

El problema de la mochila (1/2)

Dado $M \geq 0$ y dados n objetos con valores unitarios $(v_1, \ldots, v_n)$ y pesos $(p_1, \ldots, p_n)$ respectivamente, queremos calcular un vector $(x_1, \ldots, x_n)$ de valores $x_i = 0, 1$ tal que

$$\text{se maximice} \quad \sum_{i=1}^{n} x_i v_i$$

$$\text{restringido a que} \quad \sum_{i=1}^{n} x_i p_i \leq M$$

El problema de la mochila (2/2)

Hay varias opciones para la definición de los ensayos.

Una posibilidad simple es la siguiente: los ensayos son vectores de longitud $i \leq n$ con valores $0, 1$ que representan las opciones tomadas con respecto a los objetos numerados con $\{1, \ldots, i\}$; $x_i = 0$ representa la opción de no poner el objeto i en la mochila y $x_i = 1$ representa la opción de ponerlo.

Los ensayos se representan mediante un vector *X(1..n)* y una variable j; el valor de la variable j representa el identificador del objeto que debe estudiarse a continuación para extender el ensayo.

Cuando $j = n + 1$ significa que tenemos un ensayo de longitud n

Algoritmo MOCHILA_0_1

{*XOPT(1..n)* y *VOPT*: mejor solución encontrada hasta el momento y su valor, repectivamente}
{*SV* y *SP*: valor y peso, respectivamente, del ensayo en curso *X*}
{*M*: capacidad de la mochila y *V(1..n)* y *P(1..n)* los valores y pesos respectivos de los objetos}
{*j*: índice del objeto que puede extender el ensayo}

```
proc MOCHILA_0_1(SP, SV, X, j, VOPT, XOPT)
    if j = n+1 then
        if SV > VOPT then
        XOPT(1..n) ← X(1..n)
        VOPT ← SV
        end if
    else
        for k in 0..1 loop
            if SP + k·P(j) ≤ M then
                X(j) ← k
                MOCHILA_0_1(SP + X(j)·P(j), SV + X(j)·V(j), X, j+1, VOPT, XOPT)
```

La llamada inicial deberá ser MOCHILA_0_1(0, 0, X, 1, 0, XOPT)

Algoritmo alternativo a MOCHILA_0_1 (1/5)

1. En primer lugar, en vez de buscar soluciones de longitud n, consideramos solución cualquier ensayo cuyo peso no exceda la capacidad de la mochila.
2. En este caso, tenemos soluciones que son prefijos de otras soluciones.
3. Para podar ramas que ya no son prometedoras, utilizaremos un criterio que aprovecha lo que conocemos de este problema para el caso de las fracciones de objetos (véase algoritmos voraces).
4. Diremos que la extensión de un ensayo es aceptable si cabe alguna esperanza de que mejore el beneficio de la mejor carga de mochila encontrada hasta el momento.

Algoritmo alternativo a MOCHILA_0_1 (2/5)

Asumimos que $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \ldots \geq \frac{v_n}{p_n}$

Consideremos que tenemos el ensayo $(x_1, \ldots, x_i)$ con $peso = \sum_{j=1}^{i} x_j p_j$ y $valor = \sum_{j=1}^{i} x_j v_j$.

Imaginemos que podemos añadir al ensayo $(x_1, \ldots, x_i)$ los objetos completos siguientes numerados hasta el $k-1$, y que excedemos la capacidad M de la mochila si añadimos el objeto número k completo. Entonces, el peso_cota que podemos llevar en la mochila, partiendo del ensayo $(x_1, \ldots, x_i)$ es

$$peso\_cota = peso + \sum_{j=i+1}^{k-1} p_j$$

y una cota superior del beneficio sería

$$valor\_cota = (valor + \sum_{j=i+1}^{k-1} v_j) + (M - peso\_cota)\frac{v_k}{p_k}$$

Algoritmo alternativo a MOCHILA_0_1 (3/5)

Si valor_OPT es el máximo valor conseguido hasta el momento, diremos que $(x_1, \ldots, x_i)$ es aceptable si:

1. $peso \leq M$ (su peso no excede la capacidad de la mochila) y
2. $valor\_cota >$ valor_OPT (y promete más que lo conocido)

Algoritmo alternativo a MOCHILA_0_1 (4/5)

{Precondición: $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \ldots \geq \frac{v_n}{p_n}$}

```
proc MOCHILA_0_1_V(X, SP, SV, i, VOPT, XOPT)
    if SV > VOPT then
        XOPT(1..i) ← X(1..i)
        VOPT ← SV
    end if
    if i < n then
        for s in 0..1 loop
            if SP + s·P(i+1) ≤ M ∧
                  ES_ACEPTABLE(X, i+1, s, SP, SV, VOPT) then
                X(i+1) ← s
                MOCHILA_0_1_V(X, SP + s·P(i+1), SV + s·V(i+1),
                                  i+1, VOPT, XOPT)
```

La llamada inicial debe ser: MOCHILA_0_1_V(X, 0, 0, 0, 0, XOPT)

Algoritmo alternativo a MOCHILA_0_1 (5/5)

```
func ES_ACEPTABLE(X, e, X_e, SP, SV, VOPT) return boolean
    k ← e
    peso_cota ← SP + X_e·P(e)
    SV ← SV + X_e·V(e)
    while k < n ∧ peso_cota ≤ M loop
        k ← k + 1
        peso_cota ← peso_cota + P(k)
        SV ← SV + V(k)
    end loop
    if peso_cota > M then
    {k es el primer objeto que no cabe}
        peso_cota ← peso_cota - P(k)
        SV ← SV - V(k)
        valor_cota ← SV + (M - peso_cota)·V(k)/P(k)
    else   {peso_cota≤M y k=n}
        valor_cota ← SV
    end if
    return valor_cota > VOPT
```