Diseño de algoritmos

Algoritmos Voraces

Jesús Bermúdez de Andrés

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

Curso 2008-09

1. Algoritmos voraces
   - Selección de actividades
   - Problema de la mochila
   - Árbol de recubrimiento mínimo
   - Estructura de Partición
   - Caminos mínimos desde un origen

Algoritmos voraces

Resuelven problemas de optimización.
En un algoritmo voraz intervienen, típicamente, los siguientes elementos:

- Un conjunto de candidatos *Cand*. La solución se construirá con un subconjunto de *Cand*.
- Un conjunto de seleccionados S que se irán tomando del conjunto de candidatos.
- Una función *selección_voraz*(*Cand*) que escoge el candidato de *Cand* que optimiza una función según el criterio convenido (este es el elemento más característico de esta técnica de diseño de algoritmos).
- Una función *solución*(S) que determina si el conjunto de seleccionados ya forma una solución.
- Una función *completable*(X) que determina si el conjunto X puede ser completado para llegar a ser una solución; básicamente consistirá en comprobar que X satisface las restricciones del enunciado del problema.

Esquema general de un algoritmo voraz

func VORAZ(*Cand*) **return** conjunto
    $S \leftarrow \emptyset$
    while $\neg$*solución(S)* **loop**
        $x \leftarrow$ *selección_voraz*(*Cand*)
        $Cand \leftarrow Cand - \{x\}$
        if *completable*$(S \cup \{x\})$ **then** $S \leftarrow S \cup \{x\}$
    end while
    return S

Selección de actividades

Tenemos un conjunto de actividades que requieren el uso de un recurso, de manera exclusiva, durante un intervalo de tiempo.

Cada actividad está numerada. La actividad número i necesita el recurso durante el intervalo de tiempo $[c_i, f_i)$. (Naturalmente $c_i < f_i$)

Dos actividades i y j son *compatibles* si sus intervalos no se solapan (es decir, $c_i \geq f_j$ o $c_j \geq f_i$).

El problema de *selección de actividades* consiste en seleccionar un subconjunto de las actividades propuestas de manera que sean compatibles entre sí y que sea un subconjunto de cardinal máximo de entre los que forman subconjuntos compatibles.

Criterio de selección voraz

La parte fundamental de un algoritmo voraz es su criterio de *selección voraz*:

- Seleccionar la actividad que comience más temprano.
- Seleccionar la actividad de menos duración.
- Seleccionar la actividad que termine más pronto.
- Seleccionar la actividad que menos solapamientos tenga con el resto de actividades candidatas.

Para que puedan formar una solución, debemos adjuntarles el *criterio de completabilidad*:

- que no se solape con las actividades ya seleccionadas.

Demostración de corrección del criterio (1/3)

Demostración de que *"Seleccionar la actividad que termine más pronto" (que no se solape con las actividades ya seleccionadas)*
nos lleva a una solución óptima.

Demostración por inducción sobre el número de actividades seleccionadas.
Consideremos las actividades numeradas de 1 a n.

Asumimos, además, que $f_1 \leq \ldots \leq f_n$

Base de la inducción: Sea *cardinal*(S)=1. Entonces $S = \{1\}$.

Si OPT es una solución optimal cualquiera y k es la "primera" actividad de OPT (o sea $f_k = \text{mín}\{f_j : j \in OPT\}$), pueden ocurrir dos casos: $k = 1$ o $k \neq 1$.

Demostración de corrección del criterio (2/3)

Si $k = 1$, terminamos puesto que eso significaría que $S \subseteq OPT$ que es lo que queremos demostrar.

Si $k \neq 1$, obsérvese que $(OPT - \{k\}) \cup \{1\}$ es también una solución optimal ya que tiene el mismo número de actividades que OPT, y además son actividades compatibles puesto que $f_1 \leq f_k$.

Hipótesis de inducción: Sea $cardinal(S) = m - 1$ y sea S parte de una solución optimal OPT y construido según el criterio de selección indicado arriba.

Imaginemos OPT ordenado por tiempo de terminación de sus actividades.

$$OPT = \{a_1, \ldots, a_{m-1}\} \cup \{b_m, \ldots, b_n\}$$

Por el criterio de selección es obvio que las actividades de S han de ser las primeras, es decir $S = \{a_1, \ldots, a_{m-1}\}$.

Demostración de corrección del criterio (3/3)

Sea b_m la primera de $OPT - S$ (o sea, $f_{b_m} = \text{mín}\{f_j : j \in OPT - S\}$).

Sea i la actividad elegida por el criterio de selección tras haber seleccionado el conjunto S, formalmente $f_i = \text{mín}\{f_j : j$ es compatible con $S\}$.

$f_i \leq f_{b_m}$, puesto que todas las actividades de $OPT - S = \{b_m, \ldots, b_n\}$ son compatibles con S.

Entonces, $(OPT - \{b_m\}) \cup \{i\}$ es solución optimal pues tiene el mismo cardinal que OPT y la actividad i es compatible además con $(OPT - S) - \{b_m\}$.

La selección voraz m-ésima nos lleva a un conjunto de seleccionados incluido en una solución optimal.

Algoritmo Selector_Actividades

Los tiempos de comienzo y finalización respectivos de cada actividad $i = 1, \ldots, n$ están en $c(1..n)$ y $f(1..n)$

func Selector_Actividades ($c(1..n)$, $f(1..n)$)
return conjunto_de_actividades
$\{c(1..n)$ y $f(1..n)$ ordenados según el criterio $(i < j \Rightarrow f(i) \leq f(j))\}$
    $S \leftarrow \{1\}$
    $z \leftarrow 1$ $\{z$ representa a la última actividad seleccionada$\}$
    for i **in** $2 \ldots n$ **loop**
        if $c(i) \geq f(z)$ **then**
            $S \leftarrow S \cup \{i\}$
            $z \leftarrow i$
    end for
    return S

Análisis: $\Theta(n)$

Problema de la mochila

Disponemos de n artículos de valores v_i $(i : 1 \ldots n)$ y pesos p_i $(i : 1 \ldots n)$, respectivamente. Queremos cargar una mochila con fragmentos de esos artículos que maximice la suma de los valores transportados. La capacidad máxima de la mochila es $M \geq 0$.

Más formalmente, calcular un vector $(x_1, \ldots, x_n)$ de valores $0 \leq x_i \leq 1$ tal que

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

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

Hemos visto una solución a este problema (con la técnica de Programación Dinámica), para el caso en el que $\forall i : \ x_i = 0$ o 1

Criterios de selección

Criterios incorrectos (no garantizan el óptimo):

- Seleccionar, del artículo de mayor valor, el mayor fragmento que quepa en la mochila
- Seleccionar, del artículo de menor peso, el mayor fragmento que quepa en la mochila

Criterio de selección correcto:

- Seleccionar el mayor fragmento que quepa en la mochila del artículo que tenga el mayor cociente *valor/peso*

Obsérvese que, entonces, una solución optimal es de la forma siguiente $(1, \ldots, 1, x_k, 0, \ldots, 0)$ con $x_k \neq 1$, asumiendo que hemos ordenado los artículos de manera que

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

No obstante, puede haber soluciones igualmente óptimas que no estén construidas así.

Demostración de corrección de la selección voraz: (1/4)

Suponemos que $v_1/p_1 \geq \ldots \geq v_n/p_n$.
(1) Caso en el que el artículo número 1 no cabe completo en la mochila: $q_1 p_1 = M$ con $q_1 \neq 1$.
Supongamos una solución $OPT = (x_1, \ldots, x_n)$ con $x_1 < q_1$. Entonces podemos construir otra solución $OPTV = (q_1, 0, \ldots, 0)$, que transporta tanto valor como OPT, del siguiente modo:
El peso que añadimos con el aumento del fragmento del artículo 1 lo compensamos retirando los fragmentos de los otros artículos, o sea que $(q_1 - x_1)p_1 = \sum_{i=2}^{n} x_i p_i$.
Entonces el valor añadido a $OPTV$ es mayor o igual que el retirado de OPT:

$$(q_1 - x_1)v_1 = (\sum_{i=2}^{n} x_i p_i)v_1/p_1 \geq \sum_{i=2}^{n}(x_i p_i)v_i/p_i = \sum_{i=2}^{n} x_i v_i$$

y con esto queda demostrado que la selección voraz es correcta en este caso.

Demostración de corrección de la selección voraz: (2/4)

(2) Caso en el que el artículo número 1 cabe completo en la mochila.

Supongamos una solución optimal $OPT = (x_1, \ldots, x_n)$ con $x_1 \neq 1$. Entonces podemos construir otra solución optimal $OPTV$ aumentando el fragmento del artículo 1 hasta ponerlo completo y retirando un peso equivalente de fragmentos de los otros artículos.

Construimos $OPTV = (1, x'_2, \ldots, x'_n)$ de manera que $x'_i = x_i - y_i$ con $y_i \leq x_i$ donde y_i representa la porción que retiramos del artículo número i, y por tanto $\sum_{i=2}^{n} y_i p_i = (1 - x_1)p_1$ para garantizar que el $peso(OPT) = peso(OPTV) = M$.

Pero entonces se satisface lo siguiente:

Demostración de corrección de la selección voraz: (3/4)

$$
\begin{aligned}
\text{valor}(OPTV) &= v_1 + (x_2 - y_2)v_2 + \cdots + (x_n - y_n)v_n \\
&= v_1 + \sum_{i=2}^{n} x_i v_i - \sum_{i=2}^{n} y_i v_i \\
&= x_1 v_1 + (1 - x_1)v_1 + \sum_{i=2}^{n} x_i v_i - \sum_{i=2}^{n} y_i v_i \\
&\geq \text{valor}(OPT) \quad \text{esto se satisface por lo siguiente:}
\end{aligned}
$$

$$
\begin{aligned}
(1 - x_1)v_1 &= (\sum_{i=2}^{n} y_i p_i)v_1/p_1 \quad \text{por la compensación de pesos} \\
&\geq \sum_{i=2}^{n} (y_i p_i v_i / p_i) \\
&= \sum_{i=2}^{n} (y_i v_i)
\end{aligned}
$$

Demostración de corrección de la selección voraz: (4/4)

Al construir la solución optimal $OPTV = (1, x'_2, \ldots, x'_n)$ hemos reducido el problema original a otro análogo pero con una instancia más pequeña:

Ahora debemos encontrar un vector $(x_2, \ldots, x_n)$ de valores $0 \leq x_i \leq 1$ tal que

$$\text{se maximice} \quad \sum_{i=2}^{n} x_i v_i$$

$$\text{restringido a que} \quad \sum_{i=2}^{n} x_i p_i \leq M - p_1$$

y aplicando el mismo estudio que acabamos de realizar, terminamos demostrando que una solución optimal es de la forma $(1, \ldots, 1, x_k, 0, \ldots, 0)$ con $x_k \neq 1$.

Algoritmo Mochila_Frag

Por lo tanto, una solución consiste en ordenar los artículos según el criterio comentado y poner, sucesivamente, todo lo que quepa de cada artículo, tomados precisamente en ese orden.

func Mochila_Frag ($v(1..n)$, $p(1..n)$, M) **return** vector
  $peso \leftarrow 0$
  $i \leftarrow 1$ {representa el número del artículo a tratar}
  while $i \leq n \wedge peso + p(i) \leq M$ **loop**
    $x(i) \leftarrow 1$
    $peso \leftarrow peso + p(i)$
    $i \leftarrow i + 1$
  end loop
  $\{\ i < n \Rightarrow peso + p(i) > M$ y $\forall k : 1..i-1.x(k) = 1\ \}$
  if $i \leq n$ **then**
    $x(i) \leftarrow (M - peso)/p(i)$
    for k **in** $i + 1..n$ **loop** $x(k) \leftarrow 0$
  return $x(1..n)$

Árbol de recubrimiento mínimo

Dado un grafo conexo $G = (N, A)$, no dirigido, con pesos no negativos asociados a las aristas, determinar un subconjunto de aristas de A, con el menor cardinal posible, que conecte a todos los nodos de N y cuya suma de pesos sea mínima.

Obsévese que el subconjunto solución tiene que ser un árbol.

Puede haber varios árboles que sean solución.

Algoritmo de Prim

Algoritmo abstracto

func PRIM $(G = (N, A))$ **return** conjunto_de_aristas
  $ARM \leftarrow \{\ \}$ {Árbol de Recubrimiento Mínimo}
  $C \leftarrow r$ {cualquier $r \in N$, nodos Cubiertos}
  while $C \neq N$ **loop**
    "seleccionar $(a, b) \in A$ con $a \in C$ y $b \in N - C$ y de peso mínimo"
    $ARM \leftarrow ARM \cup \{(a, b)\}$
    $C \leftarrow C \cup \{b\}$
  end loop
  return ARM

Demostración de corrección del criterio de selección de Prim:

Por inducción sobre el cardinal de ARM.
Caso base: $|ARM| = 0$, es decir $ARM = \{\ \}$. Trivialmente, cualquier solución optimal incluye a ARM.

Hipótesis de inducción: $ARM = m - 1$ y existe solución optimal OPT tal que $ARM \subseteq OPT$.
Sea (a, b) la arista seleccionada en m-ésimo lugar, con $a \in C$ y $b \in N - C$:

- Si $(a, b) \in OPT$, entonces $ARM \cup \{(a, b)\} \subseteq OPT$ y hemos terminado
- Si $(a, b) \notin OPT$, entonces $OPT \cup \{(a, b)\}$ tiene un ciclo y hay dos aristas que salen de C: una es (a, b), y sea (x, y) la otra, con $x \in C$ e $y \in N - C$.
  Por el criterio de selección: $peso((a, b)) \leq peso((x, y))$.
  Entonces $OPT \cup \{(a, b)\} - \{(x, y)\}$ es una solución optimal y $ARM \cup \{(a, b)\} \subseteq OPT$.

Estructuras de datos

Calcularemos dos tablas, indexadas por los nodos del grafo:

- padre_en_C(i), al final, representará al padre del nodo i en el árbol de recubrimiento mínimo.
- Inicialmente, el único nodo que está en C es el nodo 1.
- peso(i) representará el peso de la arista (i, padre_en_C(i))
- Sucesivamente, la arista seleccionada será la correspondiente a peso(i) mínimo. Una vez añadida al árbol de recubrimiento, i pasa a estar en C y puede pasar a ser padre de algún nodo fuera de C.
- peso(i) $= -1$ representa que $i \in C$.

Algoritmo PRIM

Sea $P(1..n, 1..n)$ la matriz de adyacencia, con los pesos asociados.

```
func PRIM (P(1..n, 1..n)) return conjunto_de_aristas
   ARM ← { }
   for i in 2..n loop
      padre_en_C(i) ← 1
      peso(i) ← P(1, i)
   end loop
   for j in 1 . . . n − 1 loop   {una iteración por cada selección}
      min ← ∞
      for k in 2 . . . n loop
         if 0 ≤ peso(k) < min then
            min ← peso(k)
            s ← k
      end loop
      ARM ← ARM ∪ {(s, padre_en_C(s))}
      peso(s) ← −1
      for k in 2 . . . n loop
         if P(s, k) < peso(k) then
         peso(k) ← P(s, k)
         padre_en_C(k) ← s
      end loop
   end loop
   return ARM
```

Análisis: $\Theta(n^2)$. Con esta implementación, no depende del número de aristas.

Algoritmo de Kruskal

Algoritmo abstracto

func KRUSKAL ($G = (N, A)$) **return** conjunto_de_aristas
  $Cand \leftarrow A$
  $ARM \leftarrow \{\ \}$
  while $|ARM| \neq |N| - 1$ **loop**
    x $\leftarrow$ "la arista de menor peso de *Cand*"
    Cand $\leftarrow$ *Cand* - {x}
    if "x no forma ciclo con las aristas de *ARM*" **then**
      $ARM \leftarrow ARM \cup \{x\}$
  end loop
  return *ARM*

La demostración de corrección del criterio de selección de KRUSKAL es análoga a la de PRIM.

Estructura de Partición

Mantiene una colección de conjuntos no vacíos, disjuntos dos a dos, cuya unión forma el conjunto $\{1, \ldots, n\}$.

Cada conjunto tiene un *representante*, que típicamente es un elemento del conjunto.

Las operaciones son:

- CREA_CONJUNTO(i): crea el conjunto $\{i\}$. Puesto que los conjuntos han de ser disjuntos, se requiere que i no esté en ningún otro conjunto.
- UNIR(a, b): Dados dos representantes de conjuntos distintos, crea el conjunto unión de ambos y retira los dos conjuntos inicialmente representados por a y b.
- BUSCAR(i): devuelve el representante del conjunto al que pertenece i.

Implementación de la estructura de Partición (1/4)

La estructura de partición admite diversas representaciones.
Aquí presentamos una en la que los conjuntos son árboles, en los que la raíz es su elemento representante.

La partición está representada en un array $padre(1..n)$.
Cada elemento hace referencia a su padre en el árbol.

- $padre(x) = y$ (con $y \neq x$) significa que y es padre de x en un árbol.
- $padre(x) = x$ significa que x es raíz de un árbol.

proc CREA_CONJUNTO(i)
  $padre(i) \leftarrow i$

Implementación de la estructura de Partición (2/4)

```
func BUSCAR (i) return representante
    r ← i
    while padre(r) ≠ r loop
        r ← padre(r)
    end loop
    return r

func UNIR (a, b)
    if a < b then
        padre(b) ← a
    else
        padre(a) ← b
```

Con esta versión, un árbol de k nodos puede llegar a tener altura $k - 1$

Implementación de la estructura de Partición (3/4)

Para acotar la altura de los árboles, modificaremos UNIR para que "cuelgue" el árbol pequeño del árbol grande.

Un modo de codificar la altura h de un árbol de raíz a es aprovechar el mismo array *padre*, de manera que

$$padre(a) = -h$$

Entonces la implementación de las operaciones puede ser la siguiente:

```
proc CREA_CONJUNTO(i)
   padre(i) ← 0

func BUSCAR (i) return representante
   r ← i
   while padre(r) > 0 loop
      r ← padre(r)
   end loop
   return r
```

Implementación de la estructura de Partición (4/4)

func UNIR (a, b)
  if $padre(a) < padre(b)$ **then**
    $\{$ b tiene menos altura que a $\}$
    $padre(b) \leftarrow a$
  else
    $\{$ a tiene igual o menos altura que b $\}$
    $padre(a) \leftarrow b$
    if $padre(a) = padre(b)$ **then**
      $padre(b) \leftarrow padre(b) - 1$

Con esta versión, un árbol de k nodos tiene altura menor o igual que $\lfloor \lg k \rfloor$

Análisis de la segunda implementación

CREA_CONJUNTO() y UNIR() son $O(1)$.
BUSCAR() es $O(\mathrm{h})$ siendo h la altura del árbol de i.
El coste de m operaciones UNIR() y BUSCAR() es $O(m \log m)$.

Esta segunda implementación tiene un mejor coste global que la primera.
Pero todavía puede hacerse mejor. Véase bibliografía.

Implementación de KRUSKAL con la estructura de Partición

func KRUSKAL ($G = (N, A)$) **return** conjunto_de_aristas
  for cada $i \in N$ **loop** CREA_CONJUNTO(i) **end loop**
  $ARM \leftarrow \{\ \}$
  ORDENA(A) por peso creciente
  for cada $(a, b) \in A$ en orden creciente **loop**
    $R_a \leftarrow$ BUSCAR(a)
    $R_b \leftarrow$ BUSCAR(b)
    if $R_a \neq R_b$ **then**
      $ARM \leftarrow ARM \cup \{(a, b)\}$
      UNIR(a, b)
      if $|ARM| = |N| - 1$ **then return** ARM

Análisis de KRUSKAL

- Ordenar A es de $O(a \log a) = O(a \log n)$ (por $n - 1 \leq a \leq n^2$).
- n operaciones CREA_CONJUNTO() resulta Θ(n).
- $2a$ operaciones BUSCAR() más $(n-1)$ operaciones UNIR() resulta $O((a+n)\log(a+n)) = O(a \log n)$.

Con esta implementación, KRUSKAL$(G = (N, A))$ es $O(a \log n)$

Caminos mínimos desde un origen

Dado un grafo $G = (N, A)$ dirigido y con pesos no negativos asociados a las aristas, determinar distancias mínimas de los caminos desde un nodo origen a cada uno de los nodos restantes.

La distancia de un camino es la suma de los pesos de las aristas que forman el camino.

Algoritmo de Dijkstra

Se gestiona un conjunto de nodos seleccionados S, de manera que:

- para cualquier nodo $i \in S$: $D(i)$ es la distancia mínima de un camino desde el nodo origen a i.
- para cualquier nodo $j \notin S$: $D(j)$ es la distancia mínima de un camino *especial* desde el nodo origen a j.
  Un camino especial es un camino que parte del origen y sólo pasa por nodos de S hasta llegar a un nodo fuera de S.
- Criterio de selección voraz: seleccionar el nodo $x \notin S$ con $D(x)$ mínimo.
- Después de cada selección voraz hay que actualizar la tabla D.

Algoritmo abstracto DIJKSTRA

func DIJKSTRA ($P(1..n, 1..n)$) **return** tabla_distancias
  $Cand \leftarrow \{2, \ldots, n\}$ {implícitamente $S = N - Cand$}
  for i **in** $2 \ldots n$ **loop**
    D(i) $\leftarrow P(1, i)$
  end loop
  for j **in** $1 \ldots n - 2$ **loop**
    x $\leftarrow$ "nodo de $Cand$ con $D(x)$ mínimo"
    $Cand \leftarrow Cand - \{x\}$
    for cada $y \in Cand$ **loop**
      $D(y) \leftarrow$ **min**$(D(y), D(x) + P(x, y))$
    end loop
  end loop
  return D

Demostración de corrección del criterio de selección de Dijkstra (1/2)

Por inducción sobre el número de nodos seleccionados:

Caso base: Cuando S sólo contiene el nodo origen o, podemos asumir que $D(o) = 0$ y que $\forall i \notin S\ D(i)$ es el peso de la arista (o, i) (si existe), que es un camino especial; y si no existe tal arista, $D(i) = P(o, i) = \infty$ que sirve bien al caso.

Hipótesis de inducción:

- $\forall i \in S\ D(i)$ es la distancia mínima de un camino desde el nodo origen a i;
- $\forall j \notin S$: $D(j)$ es la distancia mínima de un camino *especial* desde el nodo origen a j; y
- $|S| = m - 1$.

Sea x el nodo seleccionado, es decir $\forall j \notin S\ D(x) \leq D(j)$.

Sea una solución óptima $OPTD$ tal que $OPTD(x) \leq D(x)$ (si fuese igual, la solución $D(x)$ sería correcta).

Demostración de corrección del criterio de selección de Dijkstra (2/2)

Entonces esa distancia $OPTD(x)$ debe conseguirse pasando por algún nodo $z \notin S$ antes de llegar x (si $OPTD(x)$ se consigue sólo pasando por nodos de S entonces, por definición de x, $D(x) \leq OPTD(x)$).

Es decir, $OPTD(x)$ se consigue sumando un camino especial a $z \notin S$ y la distancia que pueda haber desde z a x, que denominamos $\overline{d}(z, x)$.

$$OPTD(x) = D(z) + \overline{d}(z, x) \geq D(x) + \overline{d}(z, x) \geq D(x)$$

Entonces, cambiando en $OPTD$ esa distancia a x por $D(x)$ tenemos otra solución optimal $OPTD'$ tal que $OPTD'(x) = D(x)$ y $OPTD'(i) = OPTD(i)\ \forall i \neq x$.

Algoritmo Dijkstra

Consideramos el nodo origen numerado con 1.

```
func Dijkstra (P(1..n, 1..n)) return tabla
  for i in 2...n loop Cand(i) = true end loop
  for i in 2...n loop
    D(i) ← P(1, i)
  end loop
  for j in 1...n − 2 loop
    min ← ∞
    for i in 2...n loop
      if Cand(i) ∧ D(i) < min then
        min ← D(i)
        x ← i
    end loop
    Cand(x) ← false
    for i in 2...n loop
      if Cand(i) ∧ D(x) + P(x, i) < D(i) then
      D(i) ← D(x) + P(x, i)
    end loop
  end loop
  return D(1..n)
```

Análisis: $\Theta(n^2)$.

Puede hacerse otra implementación, usando un montículo para gestionar el conjunto *Cand* con sus valores $D(i)$ asociados. Si el grafo no tiene muchas aristas, puede ser más eficiente.