Algoritmos

Algoritmos voraces

Alberto Valderruten

Dept. de Computación, Universidade da Coruña

alberto.valderruten@udc.es

Contenido

Índice

Devolver el cambio (1)

- *Sistema monetario M*:
  monedas de denominación (valor) $1, 2, 5, 10, 20, 50, 100, 200$
- **Problema**: pagar *exactamente n* unidades de valor
  con un mínimo de monedas:

```
const M = [1, 2, 5, 10, 20, 50, 100, 200] {denominaciones}
función Devolver cambio (n) : conjunto de monedas
    S := conjunto vacío;                  {la solución se construye en S}
    ss := 0;                                   {suma de las monedas de S}
    mientras ss <> n hacer                {bucle voraz}
        x := mayor elemento de M : ss + x <= n;
        si no existe tal elemento entonces devolver "no hay soluciones";
        S := S U {una moneda de valor x};
        ss := ss + x;
    fin mientras;
    devolver S
fin función
```

Devolver el cambio (2)

- *¿Por qué funciona?*
  $\Rightarrow$ M adecuado y número suficiente de monedas
- **No funciona con cualquier** M:
  Ejemplo: $M = \{1, 4, 6\}$, $n = 8 \rightarrow \{6, 1, 1\}$ en vez de $\{4, 4\}$
  Este problema se resolverá con Programación Dinámica
- La función Devolver cambio es **voraz** (algoritmos ávidos, *greedy*)
  ¿Por qué voraz?
  - Selecciona el mejor *candidato* que puede en cada iteración, sin valorar consecuencias.
  - Una vez seleccionado un candidato, decide definitivamente:
    - aceptarlo, o
    - rechazarlo

    sin evaluación en profundidad de alternativas, sin retroceso...

$\rightarrow$ Algoritmos sencillos tanto en su diseño como implementación.
Cuando la técnica es adecuada, se obtienen algoritmos eficientes.

Características de los algoritmos voraces

- Resuelven **problemas de optimización**:
  En cada fase, toman una decisión (selección de un *candidato*), satisfaciendo un óptimo local según la información disponible, esperando así, en conjunto, satisfacer un óptimo global.
- Manejan un **conjunto de candidatos** C:
  En cada fase, retiran el candidato seleccionado de C, y si es aceptado se incluye en S, el conjunto donde se construye la solución $\equiv$ candidatos aceptados
- **4 funciones** (no todas aparecen explícitamente en el algoritmo):
  1. ¿S es **Solución**?
  2. ¿S es **Factible**? (¿nos lleva hacia una solución?)
  3. **Selección**: determina el mejor candidato
  4. **Objetivo**: valora S (está relacionada con *Selección*)

$\rightarrow$ Encontrar S: *Solución* que optimiza *Objetivo* (max/min)

Esquema de los algoritmos voraces

```
función Voraz (C:conjunto): conjunto
    S := conjunto vacío;          {la solución se construye en S}
    mientras C <> conjunto vacío y no solución(S) hacer {bucle voraz}
        x := seleccionar(C);
        C := C-{x};
        si factible (SU{x}) entonces S := SU{x}
    fin mientras;
    si solución (S) entonces devolver S
    sino devolver "no hay soluciones"
fin función
```

- Diseño de un algoritmo voraz:
  1. adaptar el esquema al problema
  2. introducir mejoras (ejemplo: en Devolver cambio, añadir div)
- **Problema**: Asegurarse (*demostrar*) que la técnica funciona
  No siempre funciona - ejemplo: "tomar la calle principal"

El problema de la mochila I (1)

- n objetos: $i = 1..n \begin{cases} \text{peso } w_i > 0 \\ \text{valor } v_i > 0 \end{cases}$
  Problema: cargar una *mochila* de capacidad W (unidades de peso), *maximizando el valor de su carga*.
- **Versión I**: los objetos se pueden fraccionar, y no se pierde valor
  $\equiv$ fracción $x_i, 0 \leq x_i \leq 1$
  $\Rightarrow$ el objeto i contribuye:
  en $x_i w_i$ al peso de la carga, limitado por W;
  en $x_i v_i$ al valor de la carga, que se quiere maximizar.
  $\Rightarrow \boxed{max \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i w_i \leq W}$
- + Hipótesis: $\sum_{i=1}^{n} w_i > W$, sino la solución es trivial
  $\Rightarrow$ en óptimo, $\sum_{i=1}^{n} x_i w_i = W$

El problema de la mochila I (2)

```
función Mochila 1 ( w[1..n], v[1..n], W): objetos[1..n]
    para i := 1 hasta n hacer
        x[i] := 0;              {la solución se construye en x}
    peso := 0;
    {bucle voraz:}
    mientras peso < W hacer
        i := el mejor objeto restante; {1}
        si peso+w[i] <= W entonces
            x[i] := 1;
            peso := peso+w[i]
        sino
            x[i] := (W-peso)/w[i];
            peso := W
        fin si
    fin mientras;
    devolver x
fin función
```

El problema de la mochila I (3)

Ejemplo: mochila de capacidad $W = 100$ y 5 objetos:

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	$(\sum_{i=1}^{n} w_i > W)$

¿Cuál es la función de Selección adecuada? (*sólo una es correcta!*)

1. ¿Objeto más valioso? $\leftrightarrow$ v_i max
2. ¿Objeto más ligero? $\leftrightarrow$ w_i min
3. **¿Objeto más rentable?** $\leftrightarrow$ v_i/w_i **max**

Objetos	1	2	3	4	5	Objetivo $(\sum_{i=1}^{n} x_i v_i)$
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	
v_i/w_i	2,0	1,5	2,2	1,0	1,2	
x_i (v_i max)	0	0	1	0,5	1	146
x_i (w_i min)	1	1	1	1	0	156
x_i (v_i/w_i max)	1	1	1	0	0,8	164

El problema de la mochila I (4)

- **Teorema**: Si los objetos se seleccionan **por orden decreciente de** v_i/w_i, el algoritmo Mochila 1 encuentra la solución óptima.
  Demostración por absurdo.
- **Análisis**:
  inicialización: $\Theta(n)$;
  bucle voraz: $O(1) * n$ (peor caso) $\rightarrow O(n)$
  + ordenación: $\boxed{O(nlogn)}$
- **Mejora**: con un montículo
  inicialización: $+O(n)$ (Crear montículo);
  bucle voraz: $O(logn) * n$ (peor caso) $\rightarrow \boxed{O(nlogn)}$
  $\rightarrow$ pero mejores $T(n)$
- **Ejercicio**: pseudocódigo de ambas versiones

Índice

Ordenación topológica (1)

- **Definición**:
  Ordenación de los nodos de un grafo dirigido acíclico:
  $\exists$ camino $v_i, ..., v_j \Rightarrow v_j$ aparece después de v_i
- **Aplicación**: sistema de prerrequisitos (llaves) en una titulación
  $(u, v) \equiv u$ *debe aprobarse antes* de acceder a v
  $\rightarrow$ grafo acíclico, sino la ordenación no tiene sentido
- **Observación**: La ordenación topológica no es única.
- **Definición**: Grado de entrada de $v =$ número de aristas (u, v)
- **Algoritmo**: en cada iteración, buscar *nodo de grado 0*, enviarlo a la salida y eliminarlo junto a las aristas que partan de él.
  + Hipótesis: el grafo ya está en memoria, *listas de adyacencia*
  $G = (N, A)$, $|N| = n$, $|A| = m$, $0 \leq m \leq n(n-1)$

Ordenación topológica (2)

```
función Ordenación topológica 1 (G:grafo): orden[1..n]
    Grado Entrada [1..n] := Calcular Grado Entrada (G);
    para i := 1 hasta n hacer Número Topológico [i] := 0;
    contador := 1;
    mientras contador <= n hacer
        v := Buscar nodo de grado 0 sin número topológico asignado; {*}
        si v no encontrado entonces
            devolver error "el grafo tiene un ciclo"
        sino
            Número Topológico [v] := contador;
            incrementar contador;
            para cada w adyacente a v hacer
                Grado Entrada [w] := Grado Entrada [w] - 1
        fin si
    fin mientras;
    devolver Número Topológico
fin función
```

Ordenación topológica (3)

- **Mejora**: estructura para nodos cuyo grado de entrada sea 0
  - {*} puede devolver cualquiera de ellos
  - al decrementar un grado, decidir si se incluye el nodo

$\rightarrow$ pila o *cola*

- **Ejemplo**: evolución de Grado Entrada

nodo							
1	0						
2	1	0					
3	2	1	1	1	0		
4	3	2	1	0			
5	1	1	0				
6	3	3	3	3	2	1	0
7	2	2	2	1	0		
Insertar	1	2	5	4	3,7	-	6
Eliminar	1	2	5	4	3	7	6

Ordenación topológica (4)

```
función Ordenación topológica 2 (G:grafo): orden[1..n]
    Grado Entrada [1..n] := Calcular Grado Entrada (G);
  { para i := 1 hasta n hacer Número Topológico [i] := 0; }
    Crear Cola (C); contador := 1 ;
    para cada nodo v hacer
        si Grado Entrada [v] = 0 entonces Insertar Cola (v, C);
    mientras no Cola Vacía (C) hacer
        v := Eliminar Cola (C);
        Número Topológico [v] := contador; incrementar contador;
        para cada w adyacente a v hacer
            Grado Entrada [w] := Grado Entrada [w] - 1;
            si Grado Entrada [w] = 0 entonces Insertar Cola (w, C)
        fin para
    fin mientras;
    si contador <= n entonces devolver error "el grafo tiene un ciclo"
    sino devolver Número Topológico
fin función
```

Ordenación topológica (5)

- **Análisis**: $O(n+m)$ con listas de adyacencia
  Peor caso: grafo denso $[m \to n(n-1)]$ y visita todas las aristas
  Mejor caso: grafo disperso $[m \to 0, m \to n]$

- **Ejercicios**: ¿Calcular Grado Entrada (G) es $O(n+m)$?
  Contrastar el algoritmo con la función voraz.

Índice

Árbol expandido mínimo (1)

- **a. e. m.**, árbol de expansión, árbol de recubrimiento mínimo
- Sea $G = (N, A)$ conexo, no dirigido, pesos ≥ 0 en las aristas
  Problema: T subconjunto de A tal que $G' = (N, T)$ conexo,
  peso ($\sum$ pesos de T) mínimo y $|T|$ mínimo.
- $|N| = n \Rightarrow |T| \geq n - 1$;
  pero, si $|T| > n - 1 \Rightarrow \exists$ ciclo
  $\rightarrow$ podemos quitar una arista del ciclo
  $\Rightarrow |T| = n - 1 \wedge G'$ conexo $\Rightarrow$ **árbol** (e. m.)
- **Aplicación**: instalación de cableado: ¿solución más económica?
- Técnica voraz:
  - *Candidatos*: aristas $\rightarrow S$: conjunto de aristas
  - *Solución?*: $S = T$?
  - *Factible*?: (N, S) sin ciclos (ej: S vacío es *Factible*)

Árbol expandido mínimo (2)

- **Definición**: una arista *parte* de un conjunto de nodos
  $\Leftrightarrow$ *uno* de sus extremos está en el conjunto
  (no parte $\Leftrightarrow$ sus 2 extremos están dentro/fuera del conjunto)
- **Lema**: sean $G = (N, A)$ un grafo conexo, no dirigido, pesado;
  B un subconjunto (estricto) de N;
  T un subconjunto (estricto) de A, *Factible*,
  sin aristas que partan de B;
  (u, v): la arista *más corta* que parte de B
  $\Rightarrow TU\{(u, v)\}$ es *Factible*

$\rightarrow$ Algoritmos de **Kruskal** y **Prim**

Algoritmo de Kruskal (1)

- Inicialmente: T vacío
- Invariante: (N, T) define un conjunto de *componentes conexas* (i. e. subgrafos, árboles)
- Final: sólo una componente conexa: el a. e. m.
- *Selección*: lema $\rightarrow$ **arista más corta...**
- *Factible*?: **...que una componentes conexas distintas**
- Estructuras de datos:
  - "grafo": aristas ordenadas por peso
  - árboles: Conjuntos Disjuntos (buscar(x), fusionar(A, B))

Algoritmo de Kruskal (2)

Ejemplo:

arista	(1,2)	(2,3)	(4,5)	(6,7)	(1,4)	(2,5)	(4,7)	(3,5)	...
peso	1	2	3	3	4	4	4	5	

paso	selección	componentes conexas
ini	-	1 2 3 4 5 6 7
1	(1,2)	1,2 3 4 5 6 7
2	(2,3)	1,2,3 4 5 6 7
3	(4,5)	1,2,3 4,5 6 7
4	(6,7)	1,2,3 4,5 6,7
5	(1,4)	1,2,3,4,5 6,7
6	(2,5) **rechazada**	1,2,3,4,5 6,7
7	(4,7)	1,2,3,4,5,6,7

Algoritmo de Kruskal (3)

```
función Kruskal ( G =(N,A) ) : árbol
    Ordenar A según longitudes crecientes;
    n := |N|;
    T := conjunto vacío;
    inicializar n conjuntos, cada uno con un nodo de N;
    {bucle voraz:}
    repetir
        a := (u,v) : arista más corta de A aún sin considerar;
        Conjunto U := Buscar (u);
        Conjunto V := Buscar (v);
        si Conjunto U <> Conjunto V entonces
            Fusionar (Conjunto U, Conjunto V);
            T := T U {a}
        fin si
    hasta |T| = n-1;
    devolver T
fin función
```

Algoritmo de Kruskal (4)

- **Teorema**: Kruskal calcula el árbol expandido mínimo.
  Demostración: inducción sobre $|T|$, utilizando el lema anterior
- **Análisis**: $|N| = n \wedge |A| = m$
  ordenar A: $O(m\log m) \equiv O(m\log n) : n-1 \leq m \leq n(n-1)/2$
  + inicializar n conjuntos disjuntos: $\Theta(n)$
  + $2m$ buscar (peor caso)
  y $n-1$ fusionar (siempre): $O(2m\alpha(2m,n)) = O(m\log n)$
  + resto: $O(m)$ (peor caso)
  $\Rightarrow \boxed{T(n) = O(m\log n)}$
- **Mejora**: utilizar un montículo de aristas en vez de ordenarlas
  No cambia la complejidad del peor caso pero se obtienen mejores tiempos (ejercicio).

Algoritmo de Prim (1)

- Kruskal: bosque que crece hasta convertirse en el a. e. m.
  Prim: **un único árbol**
  que va creciendo hasta alcanzar todos los nodos.
- Inicialización: $B = \{\text{nodo arbitrario}\} = \{1\}$, T vacío
- *Selección*: arista más corta que parte de B:
  $(u, v), u \in B \wedge v \in N - B$
  $\Rightarrow$ se añade (u, v) a T y v a B
- Invariante:
  T define en todo momento un a.e.m. del subgrafo (B, A)
- Final: $B = N$ (*Solución?*)

Algoritmo de Prim (2)

```
función Prim 1 ( G =(N,A) ) : árbol
    T := conjunto vacío;
    B := un nodo de N;
    mientras B <> N hacer
        a := (u,v): arista más corta que parte de B
                    (u pertenece a B y v no);
        T := T U {a};
        B := B U {v}
    fin mientras;
    devolver T
fin función
```

Algoritmo de Prim (3)

- **Ejemplo** (el mismo que para Kruskal):

paso	selección	B
ini	-	$\boxed{1}$
1	(1,2)	$\boxed{1,2}$
2	(2,3)	$\boxed{1,2,3}$
3	(1,4)	$\boxed{1,2,3,4}$
4	(4,5)	$\boxed{1,2,3,4,5}$
5	(4,7)	$\boxed{1,2,3,4,5,7}$
6	(7,6)	$\boxed{1,2,3,4,5,6,7} = N$

- **Observación**: No se producen rechazos.
- **Teorema**: Prim calcula el árbol expandido mínimo.
  Demostración: inducción sobre $|T|$, utilizando el lema anterior

Algoritmo de Prim (4)

- **Implementación**:

L: matriz de adyacencia $\equiv L[i,j] = \begin{cases} \text{distancia si } \exists(i,j) \\ \infty \text{ sino} \end{cases}$

$\rightarrow$ matriz simétrica ($\equiv$ desperdicio de memoria)

Para cada nodo $i \in N - B$:

Más Próximo $[i]$: nodo $\in B$ más próximo

Distancia Mínima $[i]$: distancia de $(i$, Más Próximo $[i])$

Para cada nodo $i \in B$:

Distancia Mínima $[i] = -1$

Algoritmo de Prim (5)

```
función Prim 2 ( L[1..n,1..n] ) : árbol
    Distancia Mínima [1] := -1;
    T := conjunto vacío;
    para i := 2 hasta n hacer
        Más Próximo [i] := 1;
        Distancia Mínima [i] := L[i,1]
    fin para;
    repetir n-1 veces:                          {bucle voraz}
        min := infinito;
        para j := 2 hasta n hacer
            si 0 <= Distancia Mínima [j] < min entonces
                min := Distancia Mínima [j];
                k := j
            fin si
        fin para;
        T := T U { ( Más Próximo [k], k ) };
        Distancia Mínima [k] := -1;             {añadir k a B}
```

Algoritmo de Prim (6)

```
        para j := 2 hasta n hacer
            si L[j,k] < Distancia Mínima [j] entonces
                Distancia Mínima [j] := L[j,k];
                Más Próximo [j] := k
            fin si
        fin para
    fin repetir;
    devolver T
fin función
```

- **Análisis**:

  inicialización $= \Theta(n)$

  bucle voraz: $n-1$ iteraciones, cada para anidado $= \Theta(n)$

  $\Rightarrow \boxed{T(n) = \Theta(n^2)}$

- ¿Posible mejora con un montículo?

  $\rightarrow O(mlogn)$, igual que Kruskal (ejercicio)

Árbol expandido mínimo (3)

- **Comparación**:

	Prim $\Theta(n^2)$	**Kruskal** $O(mlogn)$
Grafo denso: $m \to n(n-1)/2$	$\boxed{\Theta(n^2)}$	$O(n^2 logn)$
Grafo disperso: $m \to n$	$\Theta(n^2)$	$\boxed{O(nlogn)}$

Tabla: Complejidad temporal de los algoritmos de Prim y Kruskal

- **Observaciones**:
  - Existen algoritmos más eficientes, más sofisticados... más recientes (Kruskal es de 1956, Prim de 1957 [1930]).
  - Son ejemplos importantes de aplicación de la técnica voraz.
- **Ejercicios**: Contrastar Prim y Kruskal con la función voraz. Completar Prim y Kruskal para que devuelvan el peso del a. e. m.

Índice

Caminos mínimos (1)

- **Problema**: Dado un grafo $G = (N, A)$ *dirigido*,
  con longitudes en las aristas ≥ 0,
  con un nodo distinguido como *origen* de los caminos (el nodo 1):
  $\rightarrow$ *encontrar los caminos mínimos entre el nodo origen*
  y los demás nodos de N
  $\Rightarrow$ **algoritmo de Dijkstra**
- Técnica voraz:
  - 2 conjuntos de nodos:
    $$\begin{cases} S \equiv \text{seleccionados:} & \text{camino mínimo establecido} \\ C \equiv \text{candidatos:} & \text{los demás} \end{cases}$$
  - invariante: $N = S \cup C$
  - inicialmente, $S = 1 \rightarrow$ final: $S = N$: *función solución*
  - *Selección*: nodo de C con menor distancia conocida desde 1
    $\rightarrow$ existe una información *provisional* sobre distancias mínimas

Caminos mínimos (2)

- **Definición**:
  Un camino desde el origen a un nodo v es *especial*
  ssi todos sus *nodos intermedios* están en S.
- $\Rightarrow D$: vector con longitudes de caminos especiales mínimos;
  Selección de $v \leftrightarrow$ **el camino especial mínimo** $1..v$
  es *también* camino mínimo (se demuestra!).
  Al final, D contiene las longitudes de los caminos mínimos.
- **Implementación**:
  - $N = 1, 2, ..., n$ (1 es el origen)
  - $\begin{cases} L[i,j] & \geq 0 \text{ si } (i,j) \in A \\ & = \infty \text{ sino} \end{cases}$ *Matriz de adyacencia*, no simétrica

Algoritmo de Dijkstra (1)

```
función Dijkstra ( L[1..n,1..n] ) : vector[1..n]
    C := { 2, 3, ..., n};
    para i := 2 hasta n hacer
        D[i] := L[1,i];                                    {1}
    {bucle voraz:}
    repetir n-2 veces:
        v := nodo de C que minimiza D[v];
        C := C-{v};
        para cada w en C hacer
            D[w] := min ( D[w], D[v]+L[v,w] )              {2}
    fin repetir;
    devolver D
fin función
```

Algoritmo de Dijkstra (2)

- **Ejemplo**:

paso	selección	C	D[2]	D[3]	D[4]	D[5]
ini	-	$2,3,4,5$	50	30	100	$\boxed{10}$
1	5	$2,3,4$	50	30	$\boxed{20}$	10
2	4	$2,3$	40	$\boxed{30}$	20	10
3	3	2	35	30	20	10

Tabla: Evolución del conjunto C y de los caminos mínimos

- **Observación**: 3 iteraciones $= n - 2$

Algoritmo de Dijkstra (3)

- ¿Calcular también los nodos intermedios?
  $\rightarrow$ vector $P[2..n]$:
  $P[v] \equiv$ nodo que *precede* a v en el camino mínimo
  $\rightarrow$ Seguir precedentes hasta el origen

```
{1} D[i] := L[1,i]; P[i] := 1

{2} si D[w] > D[v]+L[v,w] entonces
         D[w] := D[v]+L[v,w];
         P[w] := v
    fin si
```

... y devolver P junto con D.

Algoritmo de Dijkstra (4)

- **Teorema**: Dijkstra encuentra los caminos mínimos desde el origen hacia los demás nodos del grafo.
  Demostración por inducción.
- **Análisis**: $|N| = n, |A| = m, L[1..n, 1..n]$
  Inicialización $= \Theta(n)$
  ¿Selección de v?
  $\rightarrow$ *"implementación rápida"*: recorrido sobre C
  $\equiv$ examinar $n-1, n-2, \ldots, 2$ valores en D, $\sum = \Theta(n^2)$
  Para anidado: $n-2, n-3, \ldots, 1$ iteraciones, $\sum = \Theta(n^2)$
  $$\boxed{T(n) = \Theta(n^2)}$$
- **Mejora**: si el grafo es disperso ($m << n^2$), utilizar listas de adyacencia
  $\rightarrow$ ahorro en para anidado: recorrer lista y no fila o columna de L

Algoritmo de Dijkstra (5)

- **Análisis** (Cont.):
  ¿Cómo evitar $\Omega(n^2)$ en selección?
  $\rightarrow C$: montículo min, ordenado según $D[i]$
  $\Rightarrow$ inicialización en $O(n)$
  $C := C - v$ en $O(logn)$
  Para anidado: modificar $D[w] = O(logn) \equiv$ flotar
  ¿Nº de veces? Máximo 1 vez por arista (peor caso)
  En total:
  - extraer la raíz $n-2$ veces (siempre)
  - modificar un máximo de m veces un valor de D (peor caso)
  $\Rightarrow \boxed{T(n) = O((m+n)logn)}$

  Ejercicio: escribir el pseudocódigo
- **Observación**: *"implementación rápida"* preferible si grafo denso