Análisis y diseño de algoritmos

5. Algoritmos voraces

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

23 de marzo de 2020

Índice

Problema de la Mochila continuo

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte de peso W.
- Seleccionar un conjunto de objetos de forma que:
  - no sobrepase el peso W (restricción)
  - el valor transportado sea máximo (función objetivo)
  - se permite fraccionar los objetos
- El problema se reduce a:
  - Seleccionar un subconjunto de (fracciones de) los objetos disponibles,
  - que cumpla las restricciones, y
  - que maximice la función objetivo.
- ¿Cómo resolverlo mediante un algoritmo voraz?
  - Se necesita un criterio de **selección voraz** que decida qué objeto tomar en cada momento.

Posibles criterios

- Supongamos el siguiente ejemplo:

$$W = 12 \quad w = (6, 5, 2) \quad v = (48, 35, 20) \quad v/w = (8, 7, 10)$$

Criterios	**Solución**	**Peso** W	**Valor** v
valor decreciente	$(1, 1, \frac{1}{2})$	12	93
peso creciente	$(\frac{5}{6}, 1, 1)$	12	95
valor especifico decreciente $(\frac{v_i}{w_i})$	$(1, \frac{4}{5}, 1)$	12	**96**

Formalización

- Solución: $X = (x_1, x_2, \ldots, x_n)$, $x_i \in [0, 1]$
  - $x_i = 0$: no se selecciona el objeto i
  - $0 < x_i < 1$: fracción seleccionada del objeto i
  - $x_i = 1$: se selecciona el objeto i completo
- Función objetivo:

$$\text{máx} \left(\sum_{i=1}^{n} x_i v_i \right) \qquad \text{(valor transportado)}$$

- Restricción:

$$\sum_{i=1}^{n} x_i w_i \leq W$$

algoritmo voraz (valor óptimo. v1)

```
1  double knapsack(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<unsigned> idx(w.size());     // objects sorted by value density
7      for( unsigned i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( idx.begin(), idx.end(),                  // sort by value density
10         [&v,&w]( unsigned x, unsigned y ){
11             return v[x]/w[x] > v[y]/w[y];
12         }
13     );
14     double acc_v = 0.0;
15     for( unsigned i = 0; i < idx.size(); i++ ) {
16         if( w[ idx[i] ] > W ) {
17             acc_v += W/w[ idx[i] ] * v[ idx[i] ];
18             break;
19         }
20         acc_v += v[ idx[i] ];
21         W -= w[ idx[i] ];
22     }
23     return acc_v;
24 }
```

algoritmo voraz (valor óptimo. v2)

```
1  double knapsack(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<size_t> idx(w.size());
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( idx.begin(), idx.end(),                  // sort by value density
10           [&v,&w]( size_t x, size_t y ) { return v[x]/w[x] > v[y]/w[y]; } );
11
12     double acc_v = 0.0;
13     for( auto i : idx ) {
14         if( w[i] > W ) {
15             acc_v += W/w[i] * v[i];
16             break;
17         }
18         acc_v += v[i];
19         W -= w[i];
20     }
21     return acc_v;
22 }
```

- Complejidad: $\Theta(n \log(n))$

algoritmo voraz (vector óptimo)

```
1  vector<double> knapsack_W(
2      const vector<double> &v, // values
3      const vector<double> &w, // weights
4      double W                 // knapsack weight limit
5  ){
6      vector<size_t> idx(w.size());
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8      sort( idx.begin(), idx.end(), [&v,&w]( size_t x, size_t y ){
9          return v[x]/w[x] > v[y]/w[y]; } );
10
11     vector<double> x(w.size(),0);
12     double acc_v = 0.0;
13     for( auto i : idx ) {
14         if( w[i] > W ) {
15             acc_v += W/w[i] * v[i];
16             x[i] = W/w[i];
17             break;
18         }
19         acc_v += v[i];
20         W -= w[i];
21         x[i] = 1.0;
22     }
23     return x;
24 }
```

Corrección del algoritmo /1

Teorema: El algoritmo encuentra la solución óptima

Sea $X = (x_1, x_2, \ldots, x_n)$ la solución del algoritmo $(\sum_{i=1}^{n} x_i w_i = W)$
Sea $Y = (y_1, y_2, \ldots, y_n)$ otra solución factible $(\sum_{i=1}^{n} y_i w_i = Q \leq W)$

- De la hipótesis se desprende: $W - Q = \sum_{i=1}^{n} (x_i - y_i) w_i \geq 0$
- Hay que demostrar: $V(X) - V(Y) \geq 0$, sabiendo que:
  - $V(X) - V(Y) = \sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i}$
- Sea j la posición del (único) objeto que puede estar fraccionado en X, es decir:
  - Sea $j : x_i = 1 \; \forall i < j$ y $x_i = 0 \; \forall i > j$
- Si podemos demostrar que $\boxed{(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j} \; \forall i}$,

  concluiremos:
  $V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i) w_i \frac{v_i}{w_i} \geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i = \frac{v_j}{w_j} (W - Q) \geq 0$

Corrección del algoritmo /2

Es cierto que $\boxed{(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j} \ \forall i}$?

$i < j$: (parte completamente cargada en X)

- $x_i = 1,\ y_i \leq 1 \Longrightarrow x_i - y_i \geq 0$
- $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$
- Se cumple! (por la forma en la que hemos escogido j)

$i = j$: (El elemento que puede estar fraccionado)

- $\frac{v_i}{w_i} = \frac{v_j}{w_j}$
- Se cumple! (ambas partes de la inecuación son iguales).

$i > j$: (parte completamente vacía en X)

- $x_i = 0,\ y_i \geq 0 \Longrightarrow x_i - y_i \leq 0$
- $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$
- Se cumple! (puesto que el primer factor es negativo)

Índice

Algoritmos voraces (*Greedy*)

Definición:

Un algoritmo voraz es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción local óptima en cada paso con la esperanza de llegar a una solución general óptima

Dicho de otra forma:

- Descompone el problema en un conjunto de decisiones *locales*...
- ...y elige la más prometedora
  - Es decir, aquella que se considera mejor para optimizar la medida global.
  - Pero esa decisión puede no conducir a solución óptima (depende del problema).
- Nunca reconsidera las decisiones ya tomadas.
  - Lo que conduce a algoritmos (muy) eficientes.

Algoritmos voraces: esquema

Esquema voraz

```
1  t_conjuntoElementos VORAZ(t_problema dp)
2  {
3     t_conjuntoElementos y, solucion;
4     elemento decision;
5     y=prepararDatos(dp);          // preparacion de datos para facilitar seleccion
6     while(noVacio(y) || !esSolucion(solucion)) { // quedan datos por seleccionar
7                                                  // y aun no se ha llegado a la solucion
8        decision=selecciona(y);
9        if (esFactible(decision,solucion))
10          solucion=anadeElemento(decision,solucion);
11       y=quitaElemento(decision,y);   // descartar en cualquier caso
12    }
13    return solucion;
14 }
15
```

Algoritmos voraces: esquema

Esquema voraz (recursivo)

```
1 Solution greedy( Problem p ){
2     Problem sub_prob;
3     Decision decision;
4     if( is_simple(p) )
5         return trivial(p);
6     tie(sub_prob, decision)  = select( divide(p) );
7     return Solution( greedy(sub_prob), decision );
8 }
```

Esquema voraz iterativo (iterativo)

```
1 Solucion greedy( Problem p ){
2     Problem sub_prob;
3     Decision decision;
4     Solution solution;
5     while( ! is_simple(p) )
6         solution += select( divide(p) );
7     solution += trivial(p);
8     return solution;
9  }
```

Características

- Son algoritmos eficientes y fáciles de implementar.
- Dependiendo del problema que se esté resolviendo:
  - Puede que no se encuentre la solución óptima;
  - Incluso puede que no se encuentre ninguna solución;
  - Pero si soluciona el problema entonces quizá sea la mejor solución que exista.
- Es necesario un buen criterio de selección para tener garantías de éxito.
- Aún en el caso de que no garantice solución óptima, se aplican mucho:
  - En problemas con muy alta complejidad computacional,
  - Cuando es suficiente una solución aproximada.

Índice

Problema de la mochila discreta (sin fraccionamiento)

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte peso W. Seleccionar un conjunto de objetos de forma que:
  - no sobrepase el peso W
  - el valor transportado sea máximo
- Formulación del problema:
  - Expresaremos la solución mediante un vector $(x_1, x_2, \ldots, x_n)$ donde x_i representa la decisión tomada con respecto al elemento i.
  - Función objetivo:

$$\text{máx}\left(\sum_{i=1}^{n} x_i v_i\right) \qquad \text{(valor transportado)}$$

  - Restricciones

$$\sum_{i=1}^{n} x_i w_i \leq W \qquad x_i \in \{0,1\} \begin{cases} x_i = 0 & \text{no se selecciona el objeto } i \\ x_i = 1 & \text{sí se selecciona el objeto } i \end{cases}$$

problema de la mochila discreta (sin fraccionamiento)

- En este caso el método voraz no resuelve el problema.
- Ejemplo:

$$W = 120 \quad w = (60, 60, 20) \quad v = (300, 300, 200) \quad v/w = (5, 5, 10)$$

  - solución voraz: $(0, 1, 1) \rightarrow$ valor total $= 500$
  - solución óptima: $(1, 1, 0) \rightarrow$ valor total $= 600$

$\Rightarrow$ La selección por valor específico no conduce al óptimo.

- No se conoce ningún criterio de selección (voraz o no) que conduzca al óptimo ante cualquier instancia de este problema.
- Aún así, esta solución se utiliza mucho como aproximación al óptimo (en esto consisten las heurísticas voraces).

Heurística voraz para resolver 'la mochila discreta'

```
1  double knapsack_d(
2      const vector<double> &v,
3      const vector<double> &w,
4      double W
5  ) {
6      vector<size_t> idx( w.size() );
7      for( size_t i = 0; i < idx.size(); i++) idx[i] = i;
8
9      sort( idx.begin(), idx.end(), [&w,&v]( size_t x, size_t y ){
10         return v[x]/w[x] > v[y]/w[y];
11     } );
12
13     double acc_v = 0.0;
14
15      for( auto i : idx ) {
16
17         if( w[i] < W ) {
18             acc_v += v[i];
19             W -= w[i];
20         }
21     }
22
23     return acc_v;
24 }
```

El problema del cambio

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
  - Una solución es una secuencia de decisiones

$$S = (s_1, s_2, \ldots, s_n)$$

  - La función objetivo es

$$\text{mín } |S|$$

  - La restricción es

$$\sum_{i=1}^{n} \text{valor}(s_i) = M$$

- La solución voraz es tomar en cada momento la moneda de mayor valor posible.

Ejemplo

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C:
- Sea $M = 65$

C	S	n	**Solución**
$\{1, 5, 25, 50\}$	$(50, 5, 5, 5)$	4	óptima
$\{1, 5, 7, 25, 50\}$	$(50, 7, 7, 1)$	4	óptima
	$(50, 5, 5, 5)$	4	no voraz
$\{1, 5, 11, 25, 50\}$	$(50, 11, 1, 1, 1, 1)$	6	factible pero no óptima
$\{5, 11, 25, 50\}$	$(50, 11, ?)$	???	no encuentra solución

Árbol de recubrimiento de coste mínimo

- Partimos de un grafo $g = (V, A)$:
  - conexo
  - ponderado
  - no dirigido
  - con arcos positivos
- Queremos el árbol de recubrimiento de g de coste mínimo:
  - subgrafo de g
  - con todos los vértices (recubrimiento)
  - sin ciclos (árbol)
  - conexo (árbol)
  - coste mínimo

Algoritmos de Prim y Kruskal

- Existen al menos dos algoritmos voraces que resuelven este problema,
  - algortimo de Prim
  - algortimo de Kruskal
- En ambos se van añadiendo arcos de uno en uno a la solución
- la diferencia está en la forma de elegir los arcos a añadir

Algoritmo de Prim

Algoritmo de Prim básico

1: Datos: $G = (V, A)$ ▷ Grafo no dirigido, conexo y ponderado
2: Resultado: $Sol \subseteq A$ ▷ Árbol de expansión mínimo
3: Auxiliar: $V' \subseteq V$ ▷ Conjunto de vértices visitados
4: Auxiliar: $(v, v') \in A$ ▷ Arista seleccionada
5: $Sol \leftarrow \emptyset$ ▷ Inicialmente el árbol está vacío
6: $v \leftarrow$ **elementoAleatorio**(V) ▷ Partimos de cualquier vértice de V
7: $V' \leftarrow \{v\}$ ▷ Se visita ese vértice
8: **while** $V' \neq V$ **do** ▷ Mientras no se hayan visitado todos los vértices
9: ▷ Selección: Arista de menor peso con un vértice visitado y el otro no
10: $(v, v') \leftarrow\in$ **aristaMenorPeso**(A) **con** $v \in V - V'$ **y** $v' \in V'$
11: $Sol \leftarrow Sol \cup \{(v, v')\}$ ▷ Se añade esa arista a la solución
12: $V' \leftarrow V' \cup \{v\}$ ▷ Se visita el vértice v
13: **end while**

Algoritmo de Prim

- Se mantiene un conjunto de vértices explorados
- Se coge un vértice al azar y se añade al conjunto de explorados
- en cada paso:
  - buscar el arco de mínimo peso que va de un vértice explorado a uno que no lo está
  - añadir el arco a la solución y el vértice a los explorados

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  list<edge> prim( const Graph &g ) {
2      int n = g.size();
3      vector<bool> visited( n, false);
4      list<edge> r;
5
6      edge e{-1,0};
7      for( int i = 0; i < n-1; i++ ) {
8          visited[e.d] = true;
9
10         e = min_edge( g, visited );
11
12         r.push_back(e);
13
14     }
15     return r;
16 }
```

Estructuras de datos

```
1  typedef vector<vector<int>> Graph;
2
3  struct edge {
4      int s;
5      int d;
6  };
7
8  // Graph instantiation example
9  //   (999 == \infty)
10
11 Graph g{
12    { 999,   3,   1,   6, 999, 999 },
13    {   3, 999,   5, 999,   3, 999 },
14    {   1,   5, 999,   5,   6,   4 },
15    {   6, 999,   5, 999, 999,   2 },
16    { 999,   3,   6, 999, 999,   6 },
17    { 999, 999,   4,   2,   6, 999 }
18 };
```

Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1  edge min_edge(
2      const Graph& g,
3      const vector<bool> &visited
4
5  ) {
6      int n = g.size();
7      int min = numeric_limits<int>::max();
8      edge e;
9      for( int i = 0; i < n; i++ )
10         for( int j = 0; j < n; j++ )
11             if( visited[i] && !visited[j] )
12                 if( g[i][j] < min ) {
13                     min = g[i][j];
14                     e.s = i;
15                     e.d = j;
16                 }
17
18     return e;
19 }
```

- complejidad `min-edge`: $O(V^2)$
- complejidad de `prim`: $O(V^3)$
- ¿se puede mejorar?

Algoritmo de Prim

Mejora:

- No hace falta recorrer todos los arcos cada vez
- Si cambia el mínimo es a causa del último vértice añadido
- Hay que guardarse, para cada vértice, el último mínimo
  - mediante un vector w de mínimos ya calculados que se actualiza cada vez que se visita un vértice,
  - en w_i está el peso de la mejor arista que conecta un vértice visitado con el vértice i (aún sin visitar),
  - Además, en f_i está el vértice origen de esa arista (representada por w_i)

Algoritmo de Prim

Algoritmo de Prim (con indices)

```
1  list<edge> prim( const Graph &g ) {
2      int n = g.size();
3      vector< bool > visited( n, false);                  // visited vertex
4      vector<int> w(n, numeric_limits<int>::max() );  // previous min's
5      vector<int> f(n);                               // fathe of the min's
6      list<edge> r;
7
8      edge e{-1,0};
9      for( int i = 0; i < n-1; i++ ) {
10
11         visited[e.d] = true;
12         update_idx( g, w, f, e.d );                  // update index
13
14         e = min_edge( g, w, f, visited );
15
16         r.push_back(e);
17     }
18     return r;
19 };
```

Algoritmo de Prim

Actualizar índices

```
1  void update_idx(
2      const  Graph &g,
3      vector<int>  &w,
4      vector<int>  &f,
5      int          nv
6  ) {
7
8      int n = g.size();
9      for( int j = 0; j < n; j++ )
10         if( w[j] > g[nv][j] ) {
11             w[j] = g[nv][j];
12             f[j] = nv;
13         }
14 }
```

Buscar mejor arco

```
1  edge min_edge(
2      const Graph         &g,
3      const vector<int>   &w,
4      vector<int>         &f,
5      const vector<bool> &visited
6  ) {
7      int n = g.size();
8
9      int min = numeric_limits<int>::max();
10     edge e;
11     for( int j = 0; j < n; j++ ) {
12         if( !visited[j] && w[j] < min ) {
13             min = w[j];
14             e.s = f[j];
15             e.d = j;
16         }
17     }
18     return e;
19 }
```

Algoritmo de Kruskal

Algoritmo de Kruskal básico

1: Datos: $G = (V, A)$ ▷ Grafo no dirigido, conexo y ponderado
2: Resultado: $Sol \subseteq A$ ▷ Árbol de expansión mínimo
3: Auxiliar: $A' \subseteq A$ ▷ Conjunto ordenado de aristas
4: Auxiliar: $(u, v) \in A$ ▷ Arista seleccionada
5: $Sol \leftarrow \emptyset$ ▷ Comenzamos con un bosque vacío
6: $A' \leftarrow$ **ordenarPesosCreciente**(**A**) ▷ Preparar conjunto de aristas
7: **while** $|Sol| < |V| - 1 \wedge A' \neq \emptyset$ **do**
8: $(u, v) \leftarrow$ **primero**(A') ▷ Selección: Arista de mínimo peso aún sin considerar
9: **if noCreaCiclo**$((u, v), Sol)$ **then** ▷ Ha de conectar dos árboles existentes,
10: ▷ o bien, ser un nuevo árbol (nueva componente conexa)
11: $Sol \leftarrow Sol \cup \{(u, v)\}$ ▷ Se añade a la solución
12: **end if**
13: $A' \leftarrow A' - \{(u, v)\}$ ▷ En cualquier caso se descarta la arista
14: **end while**
15: **if** $|Sol| = |V| - 1$ **then**
16: **return** Sol
17: **else**
18: **return** $\emptyset$ ▷ Grafo no conexo: no existe árbol de recubrimiento
19: **end if**

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

¿Cómo implementar la función **noCreaCiclo**$(\cdots)$?

- Una posibilidad que no sirve:
  1. Marcar cada vértice que se selecciona
  2. Una arista forma parte de la solución si sus dos vértices no están ya marcados

  - Sirve para descartar ciclos, pero ...
  - El problema es que descarta aristas que conectan dos componentes conexas
- Solución:
  - Las estructuras de conjuntos disjuntos (*Disjoint-set*)

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Conjuntos disjuntos:

- También llamado TAD unión-búsqueda (*union-find*) por las operaciones que comprende.
- Tenemos una partición de un conjunto de datos y queremos:
  - Inicializar la partición: cada elemento en un bloque distinto
  - Poder unir dos bloques de la partición (`union`)
  - Saber a qué bloque pertenece un elemento (`find`)

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Aplicándolo al algoritmo de Kruskal:

- mantener una partición de los vértices
- mientras queda más de un bloque
  - buscar el arco de menor peso que una dos bloques distintos
    - (Esto asegura que no habrán ciclos)
  - añadir el arco a la partición
  - unir los dos bloques

Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Una forma de abordarlo:

- Mediante un vector de etiquetas.
- Asignamos una etiqueta distinta a cada partición,
- `union`: reetiquetamos los elementos de uno de los bloques,
  - De manera que ambos bloques tengan la misma etiqueta (unión)
  - Complejidad $O(n)$
- `find`: consultar la etiqueta.
  - Complejidad: $O(1)$

Esta implementación básica produce una complejidad temporal del alg. de Kruskal $O(A \log A + V^2)$

- $O(A \log A)$ por la ordenación de las aristas
- $O(V^2)$ por la operación `union` para las $|V| - 1$ aristas seleccionadas
- Otras implementaciones más eficientes de conjuntos disjuntos llevan a una complejidad: $O(A \log A)$

Algoritmo de Kruskal

```
1  list<edge> kruskal( const Graph &g ) {
2      struct node { int  w; edge e; };
3
4      int n = g.size();
5      list<edge> r;
6      disjoint_set s(n);
7
8      vector<node> v;
9      for( int i = 1; i < n; i++ )
10         for( int j = 0; j < i; j++ )
11             v.push_back({ g[i][j], {i, j} });
12
13     sort( v.begin(), v.end(), []( const node &n1, const node &n2) {
14         return n1.w < n2.w;
15     });
16
17     for( auto n : v ) {
18         if( s.find(n.e.s) != s.find(n.e.d) ) {
19             r.push_back(n.e);
20             s.merge( n.e.s, n.e.d );
21         }
22     }
23     return r;
24 }
```

El fontanero diligente

- Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.
- En otras palabras, si llamamos E_i a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{i=1}^{n} E_i$$

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i.
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij}, donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j, y $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.