Práctica 3
Algoritmos Greedy

ÓSCAR BERMÚDEZ,
FRANCISCO DAVID CHARTE,
IGNACIO CORDÓN,
JOSÉ CARLOS ENTRENA,
MARIO ROMÁN
Universidad de Granada
23 de abril de 2014

Resumen

Índice

1. Terminales de venta

El problema de los terminales de venta consiste en, dada una cantidad de dinero, conseguir el mínimo número de monedas posible para devolverlo al cliente. Diseñar un algoritmo Greedy que resuelva esta situación es sencillo, dados los siguientes valores de las monedas de euro: 0.01, 0.02, 0.05, 0.10, 0.20, 0.50 y 1 euro.

1.1. Algoritmo

La técnica que se seguirá es ir escogiendo la mayor cantidad de monedas del mayor valor posible sin superar la cantidad de dinero restante a cada momento:

Algoritmo 1 Devolución en mínimo número de monedas

Entrada:

`monedas`, vector con los valores de monedas disponibles
`precio`, cantidad de dinero a devolver

1: Ordenar `monedas` de mayor a menor
2: Crear un vector `vuelta`
3: **para cada** `moneda` en `monedas`
4: `cantidad` ← `precio` / `moneda`
5: `precio` ← `precio` − `cantidad moneda`
6: Añadir [`moneda`, `cantidad`] a `vuelta`
7: **fin para**
8: **devolver** vuelta

1.2. Implementación

Implementamos el algoritmo diseñado en Ruby.

```ruby
def cambio (monedas, precio)
  vuelta = []

  monedas.sort.reverse.each { |moneda|
    numero_monedas = precio / moneda
    precio = precio - numero_monedas*moneda

    vuelta.push [moneda, numero_monedas]
```

```
  }

  return vuelta
end
```

1.3. Contraejemplo: Sellos

El buen funcionamiento del algoritmo dependerá de los valores de posibles monedas que se utilicen. En el ejemplo de los sellos de valores 0.54, 0.32, 0.17 y 0.01 euros, habrá casos en los que el algoritmo no consiga el mínimo número de monedas. Por ejemplo, si queremos obtener un valor de 0.34 euros:

- Solución del algoritmo: 0.32 + 0.01 + 0.01 (3 sellos)
- Solución óptima: 0.17 + 0.17 (2 sellos)

Esto se debe a que hay valores de sellos cuyo doble es mayor que otro valor, es decir, con dos sellos de 0.17 se consiguen 34 céntimos, que es mayor que otro sello existente superior al de 17, el sello de 32 céntimos. Esto provoca que el algoritmo escoja un sello de mayor valor de forma equivocada, ya que con dos sellos de menor valor se reduce más la distancia al objetivo, pudiendo llegar a una solución mejor.

2. Red de comunicaciones

2.1. Algoritmo

Dado un conjunto de ciudades, buscamos interconectarlas con una red que minimice la longitud de red. Modelizaremos el problema como un grafo completo G del conjunto de ciudades E. Con aristas $V = \{[a, b] |\ a, b \in C\}$, donde la arista conectando los nodos a y b tiene peso igual a la distancia que los separa:

$$\forall [a, b] \in V : \quad w([a, b]) = dist(a, b) \tag{1}$$

Pretendemos interconectar ciudades de manera que la suma total de las distancias de los caminos hechos sea mínima. Es decir, buscamos el subgrafo

recubridor de menor peso. Como un grafo recubridor con ciclos tiene siempre un subgrafo recubridor estrictamente contenido en él, buscamos sólo entre los grafos acíclicos. El subgrafo acíclico recubridor de menor costo es el árbol recubridor minimal. Es decir, buscamos el árbol recubridor minimal euclídeo de un conjunto de puntos.

Son conocidos los algoritmos de Prim y Kruskal para calcular el árbol recubridor minimal. Ambos alcanzan una complejidad temporal de $\mathcal{O}(|E|log|V|)$ usando árboles binarios y listas de adyacencia para representar el grafo.

2.2. Triangulación de Delaunay

Para reducir la carga del algoritmo, podemos reducir el grafo sobre que que buscamos el árbol generador minimal. El grafo inicial es completo, ya que toda ciudad es suceptible de ser comunicada con cualquiera otra. Pero al ser un grafo completo sobre puntos en un plano euclídeo, podemos aprovechar las propiedades de la distancia para reducir el grafo sobre el que buscamos.

Sin embargo, podemos demostrar que el árbol generador minimal está contenido en la triangulación de Delaunay; y que, por tanto, sólo es necesario aplicar el algoritmo de Kruskal o Prim al subgrafo resultante de la triangulación.

Demostración. Demostraremos que cada arista del árbol generador está contenida en la triangulación de Delaunay. Sea (p, q) una arista arbitraria del árbol generador. Consideramos el círculo que tiene como diámetro $\overline{pq}$, si hubiera otro punto r en este círculo, tendríamos:

$$\overrightarrow{pr} \leq \overrightarrow{pq} \qquad \overrightarrow{rq} \leq \overrightarrow{pq} \tag{2}$$

Y entonces podríamos formar el ciclo p, r, q, p. Sabemos que la arista de mayor peso en un ciclo no forma parte del árbol generador minimal y por tanto $\overline{pq}$ no pertenecería a él, llegando a contradicción.

Así, no puede haber ningún punto en el círculo que tiene como diámetro a $\overline{pq}$. Una arista que cumple esto está forzosamente en el diagrama de Delaunay, por aplicación de la Condición de Delaunay. □

Para encontrar la triangulación de Delaunay usar

3. Segmentación de clientes

4. Asignación de trabajos

5. Asignación de aulas

Tenemos los tiempos de comienzo y finalización de una serie de clases (s_i y $f_i, i \in \{1 \dots n\}$) y queremos encontrar el mínimo número de aulas necesarias para dar esas n clases. Observamos que el problema es equivalente a uno de coloreo de grafos, para el grafo no dirigido con función de adyacencia dada por:

$$\delta_{i,j} = \begin{cases} true & [s_i, f_i] \cap [s_j, f_j] \neq \emptyset \\ false & [s_i, f_i] \cap [s_j, f_j] = \emptyset \quad o \quad i = j \end{cases}$$

Es decir, habrá arista entre la clase $i-$ésima y la $j-$ésima si la clase $i-$ésima y la $j-$ésima no pueden impartirse en el mismo aula.

5.1. Algoritmo

Algoritmo 2 Asginación de aulas

Entrada:

`clases`, array de n objetos Clase, que almacena un tiempo de inicialización y otro de finalización

1: `adyadcentes`=$(\delta_{i,j})_{nxn}, i, j \in \{0 \dots \#clases - 1\}$
2: `sin_colorear`= $\{0 \dots \#clases - 1\}$
3: `color`=[]
4: `indice`=0
5: **mientras** `sin_colorear`$\neq \emptyset$
6: `color[indice]=[sin_colorear.pop_front]`
7: **para cada** `v`$\in$`sin_colorear`
8: **si** `adyacentes[color[indice][0]][v]` **entonces**
9: `color[indice].push(v)`
10: `sin_colorear`.delete(v)
11: **fin si**
12: **fin para**
13: **fin mientras**
14: **devolver** `#color`