Algoritmia básica

Índice

Capítulo 1

ALGORITMOS VORACES

Los algoritmos voraces se utilizan para resolver problemas de optimización: de un **conjunto de elementos candidatos**, se quiere obtener un **subconjunto solución factible** que maximice/minimice una **función objetivo**.

1. Se inicia con un conjunto vacío de candidatos.
2. Se intenta añadir el mejor elemento no escogido siguiendo una **función de selección**.
3. Si añadiendo el elemento el problema aún es **completable** (se puede llegar a la solución añadiendo mas elementos) se añade, de lo contrario se rechaza y elimina como posible elemento.
4. Si no es solución, volver al paso 2

1.1 El problema de la mochila

Un ejemplo sencillo de un problema que se puede resolver con un algoritmo voraz es el de la mochila:

- Se dispone de una **mochila** con **capacidad limitada** (C).
- Se dispone de una serie de **objetos** (n). Los objetos se pueden dividir (x_i). Además, tienen un peso asociado (p_i).
- Introducir un objeto o una fracción de este da un **beneficio**, proporcional a la cantidad de dicho objeto ($b_i x_i$).
- Se desea llenar la mochila con el **máximo beneficio** sin exceder su capacidad ($(x_1, ...x_n)$ tal que $max(\sum_{1 \leq i \leq n} b_i x_i)$ y $\sum 1 \leq i \leq n p_i x_i \leq C$).

El problema es trivial si se pueden meter todos los objetos en la mochila sin sobrecargarla. Si el peso total supera la capacidad de la mochila, la estrategia mas óptima es **tomar los objetos que proporcionen mayor beneficio por unidad de peso**.

```
/*
    Pre: Los objetos deben estar ordenados de mayor a menor según la proporción
         de beneficio obtenido por unidad de peso:
         ∀i∈1..n:peso[i]>0 ∧ ∀i∈1..n-1:benef[i]/peso[i]≥benef[i+1]/peso[i+1]
    Post: sol es solución óptima del problema de la mochila
 */
double[] mochila(double[] benef, double[] peso, double cap)
{
    // Inicialización
    double[] sol = {0};
    Capacidad resto = cap;
    int i = 1;

    // Selecciona tantos objetos como pueda
    for (i = 1; i ⩽ n && peso[i] ⩽ resto; i++)
    {
        sol[i] = 1;
        resto -= peso[i];
    }

    // Coge la fracción del último objeto que llene la mochila
    if (i ⩽ n) sol[i] = resto/peso[i]
```

```
    return sol;
}
```

Su coste temporal es lineal (o logarítmico $\Theta(n \log n)$ si hay que ordenar los vectores).

1.2 Caminos mínimos en grafos

Se desea encontrar el camino de menor peso de un grafo etiquetado con pesos no negativos para cada arista. Los grafos pueden ser dirigidos o no dirigidos.

Los grafos se pueden representar de varias maneras:

- Una **matriz de adyacencia** es una matriz cuadrada donde el elemento $[i, j]$ indica si existe una arista entre los vértices i y j (y su peso si lo tuviera).
- Una **lista de adyacencia** contiene los sucesores de un vértice i. Se tiene tantas listas como vértices en el grafo.
- Las **listas de adyacencia inversas** son similares, pero se almacenan los predecesores de cada vértice. Se puede compactar en una **lista múltiple**, donde cada lista contiene el origen y destino de una arista.

Representación	Coste espacial	Consulta existencia arista	Consulta sucesores
Matriz	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n)$
Listas	$\Theta(max(n, a))$	$\Theta(n)$	$\Theta(1)$

El **algoritmo de Dijkstra** permite encontrar el camino mínimo desde un vértice hasta el resto.

1. Inicialmente **todos los vértices están sin visitar**. El vértice origen se etiqueta con 0 y el resto se suponen a distancia infinita.
2. Calcular la **distancia mínima** de los vértices que son **vecinos del vértice actual**. Si la distancia a un vértice es menor que la etiquetada, se actualiza.
3. Si quedan vértices por visitar, se **elige el vértice cuya distancia desde el inicio sea la menor** y repetir paso 2.
4. Si se **agotan los vértices** por visitar, o son inaccesibles (todos están etiquetados con distancia infinita), se ha acabado. Alternativamente, se acaba al llegar al vértice destino (si lo hubiera).

```
/*
    Pre: g es un grafo dirigido etiquetado no negativo
    Post: D=caminosMínimos(g,v)

    Implementación adaptada de
    https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
*/
int[n] algoritmoDijkstra(int[n][n] grafo, int verticeOrigen)
{
    // Inicialización
    int[n] distancia = { -1 };
    bool[n] visitado = { false };
    distancia[verticeOrigen] = 0;

    for (int i = 1; i ⩽ n - 1; i++)
    {
        // Calcula la minima distancia (=verticeOrigen en la primera iteración)
        int actual = -1, distActual = -1;

        for (int j = 1; j ⩽ n; j++) {
            if (!visitado[j] && (!actual || distancia[j] ⩽ distActual))
            {
                actual = j; distActual = distancia[j]
            }
        }

        // Si la distancia es infinita, no hay nada mas que hacer
        if (actual == - 1) break;
```

```
        // Marca vértice consultado como visitado
        visitado[actual] = true;

        // Recalcula caminos mínimos entre vértices
        for (int vecino = 1; j ⩽ v; j++)
        {
            if (!visitado[vecino] && grafo[actual][vecino] &&
                (!distancia[vecino] || distancia[actual] +
                 grafo[actual][vecino] < distancia[vecino]))
            {
                distancia[vecino] = distancia[actual] + grafo[actual][vecino];
            }
        }
    }

    return distancia;
}
```

El algoritmo anterior tiene un coste temporal de $\theta(n^2)$. Si el grafo es disperso, se puede utilizar una versión del algoritmo que utilice listas de adyacencia y colas de prioridades. En ese caso, se puede conseguir un coste de $\theta(a \log n)$.

```
// Aristas (sucesor/peso)
typedef pair<int,int> Arista;

// Vector de listas de adyacencia (sucesores/vértice)
typedef list<Arista>[n] Grafo;

int[n] algoritmoDijkstra(Grafo grafo, int verticeOrigen) {

    // Se crea una cola de prioridades y añade nodo origen a distancia 0
    // Nota: Aquí el peso va primero. El código para ordenar con
    // segundo elemento de la pareja es mas complicado
    priority_queue<Arista, vector<Arista>, greater<Arista>> cola;
    pq.push(make_pair(0, VerticeOrigen))

    // Vector de distancias
    int[n] distancias = {-1};
    distancia[verticeOrigen] = 0;

    while (!cola.empty())
    {
        // Extrae de la cola el vértice a menor distancia
        int actual = cola.top().second();
        cola.pop();

        // Actualiza distancias
        for (auto i : grafo[actual])
        {
            int vecino = i→first, pesoArista = i→second;

            if (!distancia[vecino]
                || distancia[actual] + pesoArista < distancia[vecino])
            {
                distancia[vecino] = distancia[actual] + pesoArista;
                cola.push(make_pair(distancia[vecino], vecino))
            }
        }
    }

    return distancia;
```

}

1.3 Árboles de recubrimiento de coste mínimo

1.4 Códigos de Huffman

1.5 El problema de la selección de actividades

1.6 El problema de la minimización del tiempo de espera

1.7 Fundamentos teóricos del esquema voraz

1.8 Un problema de planificación de tareas a plazo fijo

1.9 Heurísticas voraces

1.9.1 Coloreado de grados

1.9.2 El problema del viajante de comercio

Capítulo 2

DIVIDE Y VENCERÁS

Capítulo 3

PROGRAMACIÓN DINÁMICA

Capítulo 4

BÚSQUEDA CON RETROCESO

Capítulo 5

RAMIFICACIÓN Y PODA

Capítulo 6

PROGRAMACIÓN LINEAL Y REDUCCIONES

Capítulo 7

¿¿¿¿PROBLEMAS????