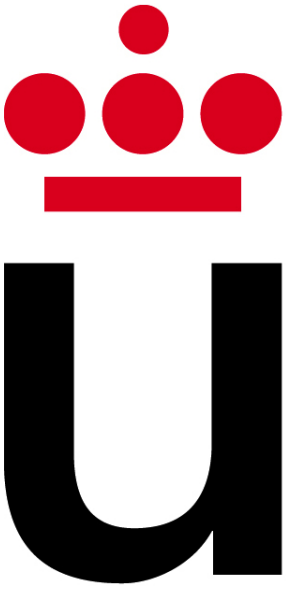
Práctica I

Simulación y Metaheurísticas

Problemas e instancias

José Ignacio Escribano



Móstoles, 8 de mayo de 2016

Índice de tablas

Índice

1. Introducción

El problema del capacited p-hub consiste en determinar los p centros (hubs) que actúan como servidores y conectar todos los clientes a uno de los servidores de forma que se minimice el la suma de las distancias de los clientes a los servidores. Además, cada servidor sólo puede puede servir una determinada cantidad de recursos, y cada cliente tiene una demanda que deben ser satisfechas. De forma matemática, el modelo se formula así:

$$\begin{aligned}
\text{Minimizar } & \sum_i \sum_j d_{ij} x_{ij} \\
\text{s.a. } & \sum_j x_{ij} = 1 \quad \forall i \\
& x_{ij} \leq y_j \quad \forall i, j \\
& \sum_j y_j = p \\
& x_{ij}, y_j \in \{0, 1\} \\
& \sum_i x_{ij} b_j \leq c
\end{aligned}$$

donde

- x_{ij} es 1 si hay arista entre el cliente i y el servidor j.
- y_i es 1 si j es servidor, 0 en caso contrario.
- d_{ij} es la distancia entre el nodo i y el j.
- b_j es la demanda del nodo j
- c es la capacidad del servidor

2. Resolución de la práctica

A continuación, resolveremos las cuestiones planteadas en la práctica.

2.1. Estructuras de datos de las instancias

Las instancias para el problema del *p-hub* se encuentran en la clase `InstanciaPHub`, cuyos atributos son los siguientes:

1. nodos: número de nodos de la instancia.
2. servidores: número de servidores de la instancia (p).
3. distancia: matriz cuadrada que contiene la distancia entre cada par de nodos, calculada a partir de las coordenadas de cada centro usando la distancia euclídea, esto es,

$$d(\mathbf{x} = (x_1, \ldots, x_n), \mathbf{y} = (y_1, \ldots, y_n)) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \quad (1)$$

4. demanda: vector con la demanda de cada nodo.

5. capacidad: capacidad de los servidores.

2.2. Carga de las instancias

Las instancias se cargan en memoria gracias al procedimiento `leerInstancia` de la clase `PHub`. Este procedimiento recibe como parámetro el nombre del fichero y va leyendo línea a línea los datos del fichero. Estos datos se guardan en sus respectivas estructuras de datos, que se pasan al constructor de la clase `InstanciaPHub`, que "genera" el objeto. Este procedimiento también se encarga de almacenar las posiciones de los centros y las almacena en memoria, para llamar a la función `calcularDistancias` de la clase `PHub`, que se encarga de devolver una matriz de distancias haciendo uso de la Ecuación 1.

2.3. Estructura de la solución

Cada solución se guarda en un objeto de la clase `Solución`, que contiene los siguientes atributos:

1. solución: vector de booleanos que indica `true`, si el nodo i es un servidor, `false` en caso contrario.

2. matrizAdyacencia: matriz de booleanos, que contiene la matriz de adyacencia del grafo, esto es, `true`, si hay una arista entre el nodo i y j, `false` en caso contrario.

3. objetivo: valor de la función objetivo, es decir, la suma de las aristas existentes entre los clientes y los objetivos.

2.4. Cálculo de la función objetivo

El cálculo de la función objetivo se realiza en la función `calcularObjetivo` de la clase `Utils`, que dada una solución, su matriz de adyacencia y de distancias, calcula su valor.
El código es el siguiente:

```
1  public static double calcularObjetivo(boolean[] solucion, boolean[][]
   ↪  matrizAdyacencia, double[][] distancia) {
2
3      double obj = 0;
4      for (int i = 0; i < matrizAdyacencia.length; i++) {
5          for (int j = 0; j <= i; j++) {
6              // Si hay conexión entre los nodos, sumamos
7              if (matrizAdyacencia[i][j]) {
8                  obj += distancia[i][j];
9              }
10         }
```

```
    }
    return obj;
}
```

Esta función recorre la matriz de adyacencia (notar que es simétrica), y si existe una arista entre el nodo i y el j, suma la distancia entre estos dos nodos a la variable `obj`, que se devuelve.

2.5. Solución aleatoria de una instancia

Para generar una solución aleatoria de una instancia está la función `generarSoluciónAleatoria` de la clase `InstanciaPHub`. El código es el siguiente:

```
public Solución generarSoluciónAleatoria() {

    Solución s1 = null;

    Random x = new Random();

    // Para poder reproducir los resultados
    x.setSeed(0);

    do{
        x = new Random();

        int nodos = this.getNodos();
        int num_servidores = 0;

        boolean[] sol = new boolean[nodos];
        boolean[][] ady = new boolean[nodos][nodos];

        // Elegimos al azar los servidores
        while (num_servidores < this.getServidores()) {
            int aleatorio = 0;

            do {
                aleatorio = x.nextInt(nodos);
            } while (sol[aleatorio]);

            sol[aleatorio] = true;

            num_servidores++;
        }

        // Generamos las aristas
        for (int i = 0; i < nodos; i++) {
            if (!sol[i]) {
                // Seleccionamos el servidor más cercano que nos encontremos
```

```
36                  int serv = Utils.seleccionarServidor(i, sol,
    ↪ this.getDistancia());
37
38                  ady[i][serv] = true;
39                  ady[serv][i] = true;
40              }
41
42          }
43
44          s1 = new Solución(sol, ady, this.getDistancia());
45
46      }while(!Utils.esSoluciónVálida(s1, this));
47
48      return s1 ;
49  }
```

En las líneas 20 a 30 se seleccionan al azar los p nodos que actuarán como servidores. Para ello, se generan números aleatorios en el intervalo discreto $[0, \text{número de nodos})$. Una vez que se tienen los nodos que actuarán como servidores, se selecciona a qué servidor se conectará cada cliente. Para ello, se busca el servidor más cercano (el que tiene una menor distancia euclídea) al cliente. De esto se encarga la función `seleccionarServidor` de la clase `Utils`. Una vez se tiene el servidor más cercano al cliente, se establece a `true` la posición de la matriz de adyacencia entre el servidor asignado y el cliente (notar que la matriz de adyacencia es simétrica). Con todos estos datos, se crea un objeto de la clase `Solución` y se comprueba si es una solución válida, es decir, si cumple las restricciones del problema. Si lo es, se devuelve la solución, y en caso contrario se repite el proceso anterior hasta que se obtenga una solución válida.

El código de la función `esSoluciónVálida` de la clase `Utils` es el siguiente:

```
1   public static boolean esSoluciónVálida(Solución s, InstanciaPHub instancia) {
2       boolean esValida = false;
3
4       boolean[][] ady = s.getMatrizAdyacencia();
5       boolean[] sol = s.getSolucion();
6       int capacidad = instancia.getCapacidad();
7
8       // Comprobamos si se cumple el criterio de la demanda
9       for (int i = 0; i < sol.length; i++) {
10          int demanda = 0;
11          for (int j = 0; j < sol.length; j++) {
12              if (sol[i] && ady[i][j]) {
13                  demanda += instancia.getDemanda()[j];
14              }
15          }
16
17          if (demanda <= capacidad && demanda != 0) {
18              esValida = true;
```

```
19          } else {
20                  esValida = false;
21          }
22      }
23      return esValida;
24  }
```

En las líneas 9 a 20 se recorre el vector solución (que contiene `true`, si el nodo i es un servidor, y `false` en caso contrario) para comprobar si se cumple la restricción de la demanda, esto es, si la suma de las demanda de cada cliente es menor o igual a la capacidad del servidor. Si se comprueba que todos los servidores tienen una demanda menor a la capacidad se devuelve `true`. En caso contrario `false`.

2.6. Programa principal

El programa principal se encarga de generar 1000 soluciones aleatorias de cada una de las instancias, y devolviendo el valor de la función objetivo y la estructura de la mejor solución de cada instancia.

Los valores de la mejor solución de cada instancia se muestran en la Tabla 1.

Tabla 1: Mejores soluciones de cada instancia

Instancia	Función objetivo	Servidores
1	768.543	[5,9,10,21,23]
2	833.446	[4,5,6,12,16]
3	802.602	[22,24,36,38,48]
4	797.988	[15,34,38,47,50]
5	709.911	[3,7,8,11,23]
6	687.306	[14,16,22,36,40]
7	851.012	[11,19, 32,41,50]
8	800.853	[4, 13, 32, 36, 48]
9	802.646	[14, 18, 28, 40, 47]
10	756.424	[10, 12, 22, 23, 38]

Notar que se han omitido la matriz de adyacencia por claridad. Las soluciones completas se pueden encontrar en el fichero `mejores_soluciones.txt` adjunto a esta memoria.

3. Conclusiones

En esta práctico nos hemos introducido en el problema p-hub. Hemos cargado instancias de este problema en memoria y hemos generado soluciones aleatorias de cada una de las instancias. En último lugar hemos simulado 1000 soluciones de cada instancia y hemos seleccionado la mejor (la que tenía menor valor de la función objetivo).