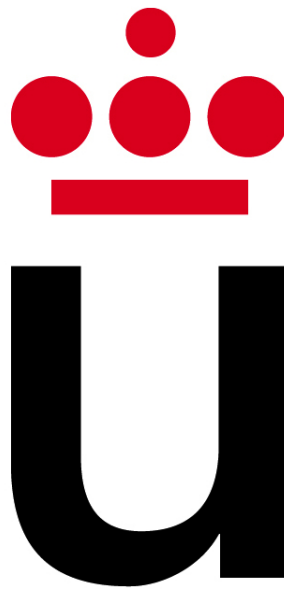


PRÁCTICA I
Simulación y Metaheurísticas
PROBLEMAS E INSTANCIAS

José Ignacio Escibano



MÓSTOLES, 21 DE MAYO DE 2016

Índice de tablas

1. Mejores soluciones de cada instancia 5

Índice

1. Introducción	1
2. Resolución de la práctica	1
2.1. Estructuras de datos de las instancias	1
2.2. Carga de las instancias	2
2.3. Estructura de la solución	2
2.4. Cálculo de la función objetivo	2
2.5. Solución aleatoria de una instancia	3
2.6. Programa principal	5
3. Conclusiones	6

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 la suma de las distancias de los clientes a los servidores. Además, cada servidor sólo 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:

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

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,

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

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 distancias 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         }

```

```

11     }
12     return obj;
13 }

```

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:

```

1  public Solución generarSoluciónAleatoria() {
2
3      Solución s1 = null;
4
5      Random x = new Random();
6
7      // Para poder reproducir los resultados
8      x.setSeed(0);
9
10     do{
11         x = new Random();
12
13         int nodos = this.getNodos();
14         int num_servidores = 0;
15
16         boolean[] sol = new boolean[nodos];
17         boolean[] [] ady = new boolean[nodos][nodos];
18
19         // Elegimos al azar los servidores
20         while (num_servidores < this.getServidores()) {
21             int aleatorio = 0;
22
23             do {
24                 aleatorio = x.nextInt(nodos);
25             } while (sol[aleatorio]);
26
27             sol[aleatorio] = true;
28
29             num_servidores++;
30         }
31
32         // Generamos las aristas
33         for (int i = 0; i < nodos; i++) {
34             if (!sol[i]) {
35                 // 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 = true;
3     boolean[][] ady = s.getMatrizAdyacencia();
4     boolean[] sol = s.getSolucion();
5     int capacidad = instancia.getCapacidad();
6     int demanda[] = new int[instancia.getServidores()];
7     int cont = 0;
8
9     // Comprobamos si se cumple el criterio de la demanda
10    for (int i = 0; i < sol.length; i++) {
11        if (sol[i]) {
12            for (int j = 0; j < sol.length; j++) {
13                if (ady[i][j]) {
14                    demanda[cont] += instancia.getDemanda()[j];
15                }
16            }
17            cont++;
18        }
19    }
20 }

```

```

19     }
20
21     for (int i = 0; i < demanda.length; i++) {
22         if (demanda[i] > capacidad) {
23             esValida = false;
24         }
25     }
26
27     return esValida;
28
29 }

```

En las líneas 9 a 19 se recorre el vector solución (que contiene **true**, si el nodo i es un servidor, y **false** en caso contrario) para calcular la demanda de ese servidor. Este proceso se repite para todos los servidores. Una vez calculado, se comprueba si existe algún servidor que no cumple la demanda, esto es, si la suma de las demandas 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	751.8302	[1, 2, 17, 19, 34]
2	855.1385	[9, 16, 20, 34, 39]
3	762.2916	[16, 22, 26, 28, 33]
4	776.3040	[26, 28, 34, 35, 41]
5	686.0421	[3, 29, 35, 43, 50]
6	689.6972	[4, 22, 29, 40, 49]
7	815.7641	[17, 19, 38, 42, 49]
8	824.0950	[4, 19, 34, 46, 49]
9	860.1325	[3, 13, 24, 37, 49]
10	736.3474	[7, 11, 22, 30, 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 práctica 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).