Maestría en Computo Estadístico
Optimización
Tarea 3
23 de mayo de 2021
Enrique Santibáñez Cortés
Repositorio de Git: Tarea 3.

Problema de la $p-$mediana

Parámetros:

- I : conjunto de instalaciones
- J : conjunto de clientes
- c_ij : costo de asignación del cliente j a la instalación i, $\forall i \in I, j \in J$
- f_i: costo de localizar la instalación $i, \forall i \in I$.
- p : número de instalaciones que se deben abrir

Variables de decisión:

$$y_i = \begin{cases} 1 & \text{si la instalación } i \text{ se abre} \\ 0 & \text{en otro caso} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{si el cliente } j \text{ se asigna a la instalacion } i \\ 0 & \text{en otro caso} \end{cases}$$

Modelo:

$$z = \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} + \sum_{i \in I} f_i y_i \tag{1}$$

$$s.a: \sum_{i \in I} y_i = p \tag{2}$$

$$\sum_{i \in I} x_{ij} = 1 \qquad \forall j \in J \tag{3}$$

$$x_{ij} \leq y_1 \qquad \forall i \in I, j \in J \tag{4}$$

$$y_i \in \{0,1\},\ x_{ij} \in \{0,1\} \qquad \forall i \in I, j \in J \tag{5}$$

$$\tag{6}$$

En la ecuación (1) la función objetivo es minimizar el costo de asignación y el costo de apertura, la restricción (2) asegura que sólo se abra la cantidad de instalaciones indicada, la restricción (3) garantiza que cada cliente sea asignado a una sola instalación, la restricción (4) asegura que cada cliente sea asignado solo a una instalación abierta y finalmente la restricción (5) es la restricción de signo de las variables de decisión.

Método Constructivo Aleatorio

Para resolver este ejercicio primero nos enfocamos en implementar el método GRASP para resolverlo, pero no pudimos implementarlo completamente. Por lo cuál, solo procedimos a realizar el algoritmo semi−greedy (ver Definición **??**) para resolver este problema. Además nos basamos en el algoritmo de GRASP aplicado en el *Problema binivel de la p-mediana con preferencia de los clientes* que se desarrollan en [**arggis**].

Definición: 1 *(Diapositiva 6, Métodos constructivos)* ***Algoritmo semi−greedy*** *Este algoritmo trata de evitar la convergencia a un mınímo local.*

Repetir hasta que la solución esté construida:

- *Para cada elemento candidato:*
  - *Aplicar una función greedy.*
  - *Ordenar los elementos de acuerdo a los valores de la función greedy.*
- *Crear una lista restringida de candidatos.*
- *Seleccionar un elemento de la lista de manera aleatoria.*
- *Añadir el elemento seleccionado a la solución.*

Considerando la **Definición ??** procedemos a describir el **Pseudocódigo ??** implementado. Basicamente consiste en una fase en cada iteración, la cual de forma iterativa agrega un elemento en cada iteración hasta que la solución alcance la cardinalidad desadad (*criterio de paro*: $|y_{optimo}| < p_k$).

Para decidir los candidatos se recurre a una función voraz que mide la contribución local de cada elemento a la solución parcial, dicha función la definimos como la suma de los costos fijos y los costos de distribución de la localización de las instalaciones ($costo\_optimo(y_{aux}, c_{ij}, f_i)$. Dado k instalaciones abiertas nosotros calculamos el costo óptimo asignando a cada cliente la instalación que tiene el costo mínimo para ese cliente, así aseguramos que estamos obteniendo la distribución de los clientes óptima para esas k instalaciones y además que estamos cumpliendo la ecuación (3) y (4).

Porteriormente, se crea una lista restrigida de candidatos (RCL) en base a los valores de dicha función. Para ello, de todos los valores obtenidos en la función calculamos el valor de c_{min} y $\alpha(c^{max} - c_{min})$. Entonces si el costo del candidato es menor que el valor calculado entonces esa instalación ingresa a la RCL.

Seleccionamos un elemento de la lista restringida de forma aleatoria ($y_{optimo} \leftarrow y_{optimo} \cup y(k)$) y actualizamos nuestra vector de candidatos ($y \leftarrow y - \{y(k)\}$). Todo el proceso se repite hasta un criterio de paro, en nuestro caso, hasta que la solución obtinene la cardinalidad p.

Este pseudocodigo se implemento en python. El archivo **codigo_EnriqueSantibanez.py** contiene las funciones construidas. La función **solution_opti** calcula los valores de nuestra función voráz definida anteriormente. Y la función **semi_greedy** implementa el pseudocodigo descrito anteriormente. El archivo **codigo_EnriqueSantibanez.ipynb** contiene los resultados de este algoritmo utilizando los datos que están en **dato_EnriqueSntibanez.txt**, además contiene una comparación con la solución óptima obtenidad con la librería *pulp*.

Algorithm 1 Semi-greedy para resolver el problema p-mediana

Input: y, c_{ij}, f_i, p_k, α

Output: y_{optimo}

$y_{optimo} \leftarrow \emptyset$
while $|y_{optimo}| < p_k$ **do**
    $costos \leftarrow \emptyset$
    for $i \in y$ **do**
        $y_{aux} \leftarrow y_{optimo} \cup i$
        $costos[i] \leftarrow costo\_optimo(y_{aux}, c_{ij}, f_i)$
    $c^{max} = \text{máx}\{costos(k) : k = 1, \cdots, len(costos)\}$
    $c_{min} = \text{mín}\{costos(k) : k = 1, \cdots, len(costos)\}$
    $RCL \leftarrow \{k = 1, \cdots, len(costos) : costos(k) \leq c_{min} + \alpha(c^{max} - c_{min})\}$
    Seleccionamos un elemento k de RCL de forma aleatoria.
    $y_{optimo} \leftarrow y_{optimo} \cup y(k)$
    $y \leftarrow y - \{y(k)\}$
return y_{optimo}