



Práctica 2.

Asignatura: Técnicas Avanzadas en Optimización

Alumnos: Ana de Souza Bossler y Jeffrey Velez Betancurth

El trabajo está orientado hacia un Problema de Localización de Plantas Simple (SPLP), buscando la forma de asignar 4 clientes a 5 plantas, de manera que podamos minimizar los costos de atención.

Se tiene información de costos $Cliente(i)-Planta(j)$, (d_{ij}) en **miles** de euros,

| CLIENTES | PLANTAS | | | | |
|----------|---------|------|------|------|------|
| | 1 | 2 | 3 | 4 | 5 |
| 1 | 7000 | 4000 | 2000 | 1000 | 5000 |
| 2 | 5000 | 6000 | 7000 | 8000 | 3000 |
| 3 | 4000 | 3000 | 2000 | 1000 | 6000 |
| 4 | 8000 | 7000 | 6000 | 5000 | 4000 |

Y de los costos fijos, de poder abrir alguna de las plantas,

| COSTO | PLANTAS | | | | |
|-------|---------|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 |
| D_j | 10 | 20 | 30 | 40 | 50 |

Nuestra función objetivo, que queremos minimizar, es la suma de los costos de asignación de los clientes a las plantas y los costos fijos de apertura de las plantas.

La función objetivo del modelo será,

$$Min: \sum_{i=1}^4 \sum_{j=1}^5 d_{ij}x_{ij} + \sum_{j=1}^5 D_j y_j$$

Queremos asegurarnos de que cada cliente sea atendido por exactamente una planta, por lo que introducimos las siguientes restricciones:

$$x_{11} + x_{12} + x_{13} + x_{14} + x_{15} = 1$$

$$x_{21} + x_{22} + x_{23} + x_{24} + x_{25} = 1$$

$$x_{31} + x_{32} + x_{33} + x_{34} + x_{35} = 1 \quad x_{41} + x_{42} + x_{43} + x_{44} + x_{45} = 1$$

También queremos asegurarnos de que un cliente sólo puede ser atendido por una planta si está abierta, introduciendo la restricción

$x_{ij} \leq y_j$ para cada i y j.

Entonces, la función objetivo del modelo queda sujeta a las siguientes restricciones,

$$\sum_{j=1}^5 x_{ij} = 1; \quad i = 1,2,3,4$$

$$x_{ij} \leq y_j; \quad i = 1,2,3,4; \quad j = 1,2,3,4,5$$

$$x_{ij} \in \{0,1\}; \quad y_j \in \{0,1\}; \quad i = 1,2,3,4; \quad j = 1,2,3,4,5$$

La variable $y_j = \begin{cases} 1; & \text{planta } j \text{ está abierta} \\ 0; & \text{en caso contrario} \end{cases}$

La variable $x_{ij} = \begin{cases} 1; & \text{si el cliente } i \text{ es atendido por la planta } j \\ 0; & \text{en caso contrario} \end{cases}$

Programamos en pulp :

```
❶ from pulp import *
# Definir el modelo
prob = LpProblem("SPLP", LpMinimize)

# Conjuntos y parámetros
clientes = [1, 2, 3, 4]
plantas = [1, 2, 3, 4, 5]
costos = {(1,1): 7000, (1,2): 4000, (1,3): 2000, (1,4): 1000, (1,5): 5000,
           (2,1): 5000, (2,2): 6000, (2,3): 7000, (2,4): 8000, (2,5): 3000,
           (3,1): 4000, (3,2): 3000, (3,3): 2000, (3,4): 1000, (3,5): 6000,
           (4,1): 8000, (4,2): 7000, (4,3): 6000, (4,4): 5000, (4,5): 4000}
costo_fijo = {1: 10, 2: 20, 3: 30, 4: 40, 5: 50}

# Variables
x = LpVariable.dicts("x", ((i, j) for i in clientes for j in plantas), lowBound=0, upBound=1, cat='Binary')
y = LpVariable.dicts("y", plantas, lowBound=0, upBound=1, cat='Binary')

# Restricciones
for i in clientes:
    prob += lpSum(x[i,j] for j in plantas) == 1, f"Suma_Clientes_{i}"
for i in clientes:
    for j in plantas:
        prob += x[i,j] <= y[j], f"Planta_{j}_Abierta_Para_Cliente_{i}"

# Función objetivo
prob += lpSum(costos[i,j]*x[i,j] for i in clientes for j in plantas) + lpSum(costo_fijo[j]*y[j] for j in plantas), "Z"

# Resolviendo el modelo
prob.solve()

# Imprimir la solución
print("Status:", LpStatus[prob.status])
for v in prob.variables():
    print(v.name, "=", v.varValue)
print("Objective value:", value(prob.objective))
```

```

Status: Optimal
x_(1,_1) = 0.0
x_(1,_2) = 0.0
x_(1,_3) = 0.0
x_(1,_4) = 1.0
x_(1,_5) = 0.0
x_(2,_1) = 0.0
x_(2,_2) = 0.0
x_(2,_3) = 0.0
x_(2,_4) = 0.0
x_(2,_5) = 1.0
x_(3,_1) = 0.0
x_(3,_2) = 0.0
x_(3,_3) = 0.0
x_(3,_4) = 1.0
x_(3,_5) = 0.0
x_(4,_1) = 0.0
x_(4,_2) = 0.0
x_(4,_3) = 0.0
x_(4,_4) = 0.0
x_(4,_5) = 1.0
y_1 = 0.0
y_2 = 0.0
y_3 = 0.0
y_4 = 1.0
y_5 = 1.0
Objective value: 9090.0

```

Podemos ver que abren las plantas 4 y 5. Los clientes 1 y 3 son atendidos por las plantas 4 y los clientes 2 y 4 por la planta 5. El valor de la función objetivo es 9090.

Ahora relajamos la restricción de que sólo pueda tomar valores enteros- se cambia la categoría de las variables en la definición “Variables” de binary para continuous:

```

❷ from pulp import *
# Definir el modelo
prob = LpProblem("SPLP", LpMinimize)

# Conjuntos y parámetros
clientes = [1, 2, 3, 4]
plantas = [1, 2, 3, 4, 5]
costos = {(1,1): 7000, (1,2): 4000, (1,3): 2000, (1,4): 1000, (1,5): 5000,
           (2,1): 5000, (2,2): 6000, (2,3): 7000, (2,4): 8000, (2,5): 3000,
           (3,1): 4000, (3,2): 3000, (3,3): 2000, (3,4): 1000, (3,5): 6000,
           (4,1): 8000, (4,2): 7000, (4,3): 6000, (4,4): 5000, (4,5): 4000}
costo_fijo = [1: 10, 2: 20, 3: 30, 4: 40, 5: 50]

# Variables
x = LpVariable.dicts("x", ((i, j) for i in clientes for j in plantas), lowBound=0, upBound=1, cat='Continuous')
y = LpVariable.dicts("y", plantas, lowBound=0, upBound=1, cat='Continuous')

# Restricciones
for i in clientes:
    prob += lpSum(x[i,j] for j in plantas) == 1, f"Suma_Clientes_{(i)}"

for i in clientes:
    for j in plantas:
        prob += x[i,j] <= y[j], f"Planta_{(j)}_Abierta_Para_Cliente_{(i)}"

# Función objetivo
prob += lpSum(costos[i,j]*x[i,j] for i in clientes for j in plantas) + lpSum(costo_fijo[j]*y[j] for j in plantas), "Z"

# Resolviendo el modelo
prob.solve()

# Imprimir la solución
print("Status:", LpStatus[prob.status])
for v in prob.variables():
    print(v.name, "=", v.varValue)
print("Objective value:", value(prob.objective))

```

```

❸ Status: Optimal
x_(1,_1) = 0.0
x_(1,_2) = 0.0
x_(1,_3) = 0.0
x_(1,_4) = 1.0
x_(1,_5) = 0.0
x_(2,_1) = 0.0
x_(2,_2) = 0.0
x_(2,_3) = 0.0
x_(2,_4) = 0.0
x_(2,_5) = 1.0
x_(3,_1) = 0.0
x_(3,_2) = 0.0
x_(3,_3) = 0.0
x_(3,_4) = 1.0
x_(3,_5) = 0.0
x_(4,_1) = 0.0
x_(4,_2) = 0.0
x_(4,_3) = 0.0
x_(4,_4) = 0.0
x_(4,_5) = 1.0
y_1 = 0.0
y_2 = 0.0
y_3 = 0.0
y_4 = 1.0
y_5 = 1.0
Objective value: 9090.0

```

Vemos que el resultado no cambia y calculamos el gap: como tienen el mismo valor, el gap es =0.

Además, introducimos una restricción de "clic" que establece que si la planta 1 está abierta, entonces la planta 5 debe estar cerrada y viceversa. Esto se traduce como $y_1 + y_5 \leq 1$. El resultado de la distribución de clientes en las plantas y del valor de la función objetivo no cambian:

```

from pulp import *

# Definir el modelo
prob = LpProblem("SPLP", LpMinimize)

# Conjuntos y parámetros
clientes = [1, 2, 3, 4]
plantas = [1, 2, 3, 4, 5]
costos = {(1,1): 7000, (1,2): 4000, (1,3): 2000, (1,4): 1000, (1,5): 5000,
           (2,1): 5000, (2,2): 6000, (2,3): 7000, (2,4): 8000, (2,5): 3000,
           (3,1): 4000, (3,2): 3000, (3,3): 2000, (3,4): 1000, (3,5): 6000,
           (4,1): 8000, (4,2): 7000, (4,3): 6000, (4,4): 5000, (4,5): 4000}
costo_fijo = [1: 10, 2: 20, 3: 30, 4: 40, 5: 50]

# Variables
x = LpVariable.dicts("x", ((i, j) for i in clientes for j in plantas), lowBound=0, upBound=1, cat='Binary')
y = LpVariable.dicts("y", plantas, lowBound=0, upBound=1, cat='Binary')

# Restricciones
for i in clientes:
    prob += lpSum(x[i,j] for j in plantas) == 1, f"Suma_Clientes_{i}"
for i in clientes:
    for j in plantas:
        prob += x[i,j] <= y[j], f"Planta_{j}_Abierta_Para_Cliente_{i}"
# Restricción del "clic"
prob += y[1] + y[5] <= 1, "Restriccion_Clic"

# Función objetivo
prob += lpSum(costos[i,j]*x[i,j] for i in clientes for j in plantas) + lpSum(costo_fijo[j]*y[j] for j in plantas), "Z"

# Resolviendo el modelo
prob.solve()

# Imprimir la solución
print("Status:", LpStatus[prob.status])
for v in prob.variables():
    print(v.name, "=", v.varValue)
print("Objective value:", value(prob.objective))

```

```

Status: Optimal
x_(1,_1) = 0.0
x_(1,_2) = 0.0
x_(1,_3) = 1.0
x_(1,_4) = 1.0
x_(1,_5) = 0.0
x_(2,_1) = 0.0
x_(2,_2) = 0.0
x_(2,_3) = 0.0
x_(2,_4) = 0.0
x_(2,_5) = 1.0
x_(3,_1) = 0.0
x_(3,_2) = 0.0
x_(3,_3) = 0.0
x_(3,_4) = 1.0
x_(3,_5) = 0.0
x_(4,_1) = 0.0
x_(4,_2) = 0.0
x_(4,_3) = 0.0
x_(4,_4) = 0.0
x_(4,_5) = 1.0
y_1 = 0.0
y_2 = 0.0
y_3 = 0.0
y_4 = 1.0
y_5 = 1.0
Objective value: 9090.0

```

Finalmente, introducimos una restricción de "agujero impar", que se traduce como

$$x_{11} + y_1 + x_{12} + x_{21} + x_{23} + y_3 + x_{45} \leq 6 - 1/3.$$

Esta restricción, añadida al clic, tampoco cambia el resultado:

```

❸ # Definir el modelo
prob = LpProblem("SPLP", LpMinimize)

# Conjuntos y parámetros
clientes = [1, 2, 3, 4]
plantas = [1, 2, 3, 4, 5]
costos = {(1,1): 7000, (1,2): 4000, (1,3): 2000, (1,4): 1000, (1,5): 5000,
           (2,1): 5000, (2,2): 6000, (2,3): 7000, (2,4): 8000, (2,5): 3000,
           (3,1): 4000, (3,2): 3000, (3,3): 2000, (3,4): 1000, (3,5): 6000,
           (4,1): 8000, (4,2): 7000, (4,3): 6000, (4,4): 5000, (4,5): 4000}
costo_fijo = {1: 10, 2: 20, 3: 30, 4: 40, 5: 50}

# Variables
x = LpVariable.dicts("x", ((i, j) for i in clientes for j in plantas), lowBound=0, upBound=1, cat='Binary')
y = LpVariable.dicts("y", plantas, lowBound=0, upBound=1, cat='Binary')

# Restricciones
for i in clientes:
    prob += lpSum(x[i,j] for j in plantas) == 1, f"Suma_Clientes_{i}"

for i in clientes:
    for j in plantas:
        prob += x[i,j] <= y[j], f"Planta_{j}_Abierta_Para_Cliente_{i}"

# Restricción del "clic"
prob += y[1] + y[5] <= 1, "Restriccion_Clic"

# Restricción de agujero impar
prob += x[1,1] + x[1,2] + x[2,1] + x[2,3] + y[3] + x[4,5] <= 6 - 1/3, "Restriccion_Agujero_Impar"

# Función objetivo
prob += lpSum(costos[i,j]*x[i,j] for i in clientes for j in plantas) + lpSum(costo_fijo[j]*y[j] for j in plantas), "Z"

# Resolviendo el modelo
prob.solve()

# Imprimir la solución
print("Status:", LpStatus[prob.status])
for v in prob.variables():
    print(v.name, "=", v.varValue)
print("Objective value:", value(prob.objective))

```

```

Status: Optimal
x_(1,-1) = 0.0
x_(1,-2) = 0.0
x_(1,-3) = 0.0
x_(1,-4) = 1.0
x_(1,-5) = 0.0
x_(2,-1) = 0.0
x_(2,-2) = 0.0
x_(2,-3) = 0.0
x_(2,-4) = 0.0
x_(2,-5) = 1.0
x_(3,-1) = 0.0
x_(3,-2) = 0.0
x_(3,-3) = 0.0
x_(3,-4) = 1.0
x_(3,-5) = 0.0
x_(4,-1) = 0.0
x_(4,-2) = 0.0
x_(4,-3) = 0.0
x_(4,-4) = 0.0
x_(4,-5) = 1.0
y_1 = 0.0
y_2 = 0.0
y_3 = 0.0
y_4 = 1.0
y_5 = 1.0
Objective value: 9090.0

```

El problema puede ser definido como:

MODELO

$$\text{Minimizar: } \sum_{i=1}^4 \sum_{j=1}^5 d_{ij} x_{ij} + \sum_{j=1}^5 D_j y_j$$

Sujeto a:

$$\sum_{j=1}^5 x_{ij} = 1; \quad i = 1,2,3,4$$

$$x_{ij} \leq y_j; \quad i = 1,2,3,4; \quad j = 1,2,3,4,5$$

$$x_{ij} \in \{0,1\}; \quad y_j \in \{0,1\}; \quad i = 1,2,3,4; \quad j = 1,2,3,4,5$$

También hemos programado una versión “más compleja” pero no hemos descrito el modelo: Imaginamos 49 clientes, que tienen que ser servidos por hasta 7 plantas, que tienen un coste fijo de abrir de 500 euros por planta y cada km viajado cuesta 3 euros. Se simula las ubicaciones de los clientes y de las plantas.

```
[ ] import numpy as np
import pulp
from itertools import combinations

# Parámetros del problema
num_clients = 49
num_branches = 7
fixed_cost = 500
travel_cost = 3
np.random.seed(0) # Para reproducibilidad

# Generar ubicaciones de clientes y sucursales aleatoriamente
client_locs = np.random.rand(num_clients, 2)
branch_locs = np.random.rand(num_branches, 2)

# Calcular distancias entre cada cliente y cada sucursal
distances = np.sqrt(((client_locs[:, np.newaxis, :] - branch_locs[np.newaxis, :, :])**2).sum(axis=2))

# Crear un problema de programación lineal
problem = pulp.LpProblem("FacilityLocation", pulp.LpMinimize)

# Crear variables binarias para cada sucursal y cada par de sucursal-cliente
branch_vars = pulp.LpVariable.dicts("Branch", range(num_branches), cat='Binary')
client_vars = pulp.LpVariable.dicts("Client", [(i, j) for i in range(num_clients) for j in range(num_branches)], cat='Binary')

# Función objetivo: minimizar el coste total de abrir sucursales y servir a los clientes
problem += pulp.lpSum(fixed_cost * branch_vars[j] + pulp.lpSum(travel_cost * distances[i][j] * client_vars[(i, j)] for i in range(num_clients) for j in range(num_branches)))

# Restricciones: cada cliente es servido por exactamente una sucursal
for i in range(num_clients):
    problem += pulp.lpSum(client_vars[(i, j)] for j in range(num_branches)) == 1

# Restricciones: una sucursal solo puede servir a un cliente si está abierta
for i in range(num_clients):
    for j in range(num_branches):
        problem += client_vars[(i, j)] <= branch_vars[j]

# Restricciones: cada cliente es servido por exactamente una sucursal
for i in range(num_clients):
    problem += pulp.lpSum(client_vars[(i, j)] for j in range(num_branches)) == 1

# Restricciones: una sucursal solo puede servir a un cliente si está abierta
for i in range(num_clients):
    for j in range(num_branches):
        problem += client_vars[(i, j)] <= branch_vars[j]

# Restricciones: cada sucursal sirve a un número par de clientes (clic)
for j in range(num_branches):
    problem += 2 * branch_vars[j] <= pulp.lpSum(client_vars[(i, j)] for i in range(num_clients))
    problem += pulp.lpSum(client_vars[(i, j)] for i in range(num_clients)) <= 2 * (num_clients + 1) * branch_vars[j]

# Añadir las restricciones para asegurar un "agujero impar"
combinations_vars = pulp.LpVariable.dicts("Combination", range(len(list(combinations(range(num_branches), 5)))), cat='Binary')
for k, branches in enumerate(combinations(range(num_branches), 5)):
    # Añadir restricción para que al menos una combinación de 5 sucursales esté abierta
    problem += pulp.lpSum(branch_vars[j] for j in branches) >= 5 * combinations_vars[k]
    problem += pulp.lpSum(combinations_vars[k] for k in range(len(list(combinations(range(num_branches), 5))))) >= 1

# Resolver el problema
problem.solve()

# Comprobar si se ha encontrado una solución válida
if pulp.LpStatus[problem.status] == 'Optimal':
    # Si se ha encontrado una solución, imprimir el resultado y terminar el bucle
    print("Estado:", pulp.LpStatus[problem.status])

print("Coste mínimo:", pulp.value(problem.objective))

opened_branches = [j for j in range(num_branches) if pulp.value(branch_vars[j]) > 0.5]
print("Sucursales abiertas:", opened_branches)

assigned_clients = {(i, j): pulp.value(client_vars[(i, j)]) for i in range(num_clients) for j in range(num_branches) if pulp.value(client_vars[(i, j)]) > 0.5}
print("Asignaciones de clientes:", assigned_clients)
```

Estado: Optimal
Coste mínimo: 2533.000216299608
Sucursales abiertas: [0, 1, 3, 4, 6]
Asignaciones de clientes: {()

Ya si prohibirramos como restricción que se formara un agujero impar, tendríamos 4 empresas abiertas y un coste de aprox. 20% del con agujero:

```

] import numpy as np
import pulp
from itertools import combinations

# Parámetros del problema
num_clients = 49
num_branches = 7
fixed_cost = 500
travel_cost = 3
np.random.seed(0) # Para reproducibilidad

# Generar ubicaciones de clientes y sucursales aleatoriamente
client_locs = np.random.rand(num_clients, 2)
branch_locs = np.random.rand(num_branches, 2)

# Calcular distancias entre cada cliente y cada sucursal
distances = np.sqrt(((client_locs[:, np.newaxis, :] - branch_locs[np.newaxis, :, :])**2).sum(axis=2))

# Crear un problema de programación lineal
problem = pulp.LpProblem("FacilityLocation", pulp.LpMinimize)

# Crear variables binarias para cada sucursal y cada par de sucursal-cliente
branch_vars = pulp.LpVariable.dicts("Branch", range(num_branches), cat='Binary')
client_vars = pulp.LpVariable.dicts("Client", [(i, j) for i in range(num_clients) for j in range(num_branches)], cat='Binary')

# Función objetivo: minimizar el costo total de abrir sucursales y servir a los clientes
problem += pulp.lpSum(fixed_cost * branch_vars[j] for j in range(num_branches)) + pulp.lpSum(travel_cost * distances[i][j] * client_vars[(i, j)] for i in range(num_clients) for j in range(num_branches))

# Restricciones: cada cliente es servido por exactamente una sucursal
for i in range(num_clients):
    problem += pulp.lpSum(client_vars[(i, j)] for j in range(num_branches)) == 1

# Restricciones: una sucursal solo puede servir a un cliente si está abierta
for i in range(num_clients):
    for j in range(num_branches):
        problem += client_vars[(i, j)] <= branch_vars[j]

# Restricciones: cada sucursal sirve a un número par de clientes (clic)
for j in range(num_branches):
    problem += 2 * branch_vars[j] <= pulp.lpSum(client_vars[(i, j)] for i in range(num_clients))
    problem += pulp.lpSum(client_vars[(i, j)] for i in range(num_clients)) <= 2 * (num_clients + 1) * branch_vars[j]

# Restricciones de agujero impar: no se puede seleccionar un conjunto de sucursales que formen un agujero impar
for branches in combinations(range(num_branches), 5):
    problem += pulp.lpSum(branch_vars[j] for j in branches) <= 4

# Resolver el problema
problem.solve()

# Imprimir el resultado
print("Estado:", pulp.LpStatus[problem.status])
print("Coste mínimo:", pulp.value(problem.objective))

opened_branches = [j for j in range(num_branches) if pulp.value(branch_vars[j]) > 0.5]
print("Sucursales abiertas:", opened_branches)

assigned_clients = {(i, j): pulp.value(client_vars[(i, j)]) for i in range(num_clients) for j in range(num_branches) if pulp.value(client_vars[(i, j)]) > 0}
print("Asignaciones de clientes:", assigned_clients)

```

Estado: Optimal
Coste mínimo: 558.470084985377
Sucursales abiertas: [4]
Asignaciones de clientes: {(0, 4): 1.0, (1, 4): 1.0, (2, 4): 1.0, (3, 4): 1.0, (4, 4): 1.0, (5, 4): 1.0, (6, 4): 1.0, (7, 4): 1.0, (8, 4): 1.0, (9, 4): 1.0, (10, 4): 1.0, (11, 4): 1.0, (12, 4): 1.0, (13,