IE 501 Project

Dattaraj Salunkhe (22B1296) & Swayam Patel (22B1816)

26^{th} November, 2024

Abstract

In this project, we delve into the classical Capacitated Facility Location Problem (CFLP), focusing on its non-deterministic optimization characteristics. We analyze the non-deterministic optimization technique to address the inherent complexities of the CFLP effectively. Building on this foundation, we extend our study to explore Competitive Facility Location Problem (CoFLP) where multiple conflicting objectives are considered, such as cost minimization, service quality, and competitive positioning. This progression aims to provide deeper insights into both classical and competitive facility location problems, contributing to the development of robust and versatile solutions.

Introduction

The Facility Location Problem (FLP) is a pivotal optimization challenge in operations research and supply chain management, focusing on determining optimal locations for facilities to minimize costs or maximize service efficiency. This problem involves balancing fixed facility costs, transportation expenses, and service delivery effectiveness. FLP finds applications in diverse areas such as logistics, urban planning, telecommunication networks, and healthcare services.

The problem is typically classified into several types,amongst which one of the most popular is *capacitated facility location problem* (CFLP), which incorporates constraints on facility capacities. Mathematical models such as integer programming and heuristic algorithms, including genetic algorithms and simulated annealing, are often employed to solve these problems.

Classical CFLP

The Capacitated Facility Location Problem (CFLP) is a variant of the classical Facility Location Problem (FLP) that incorporates capacity constraints on the facilities. In CFLP, the objective is to determine the optimal locations for facilities and the allocation of clients to these facilities

such that the total cost is minimized while ensuring that the demand assigned to each facility does not exceed its capacity.
CFLP involves two primary cost components:

1. **Facility Opening Costs:** The fixed costs associated with establishing and operating a facility at a given location.
2. **Transportation Costs:** The variable costs incurred in meeting client demands, typically based on distance or the volume of goods transported.

Our Implementation of classical CFLP

Summary

We start by defining a graph and applying Dijkstra's algorithm to compute the shortest paths between two points. In subsequent steps, we use these shortest distances directly. Next, we define capacities and demands for each location as parameters. Two decision variables are introduced: x_{ij}, indicating whether facility j serves location i, and y_i, indicating whether location i becomes a facility. The objective function minimizes the total distance required to serve all locations efficiently.

Graph Definition and Shortest Paths

A graph $G = (V, E)$ is defined, where V represents the set of nodes (locations) and E represents the set of edges (connections). Dijkstra's algorithm is used to compute the shortest path d_{ij} between any two nodes i and j. These distances d_{ij} are used in further computations.

Python Code: Shortest Paths

```
import networkx as nx

# Create a graph and calculate shortest paths
G = nx.Graph()
G.add_edge(1, 2, weight=4)
G.add_edge(2, 3, weight=2)
G.add_edge(3, 4, weight=5)

# Compute shortest paths
shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G, weight='weight'))
print(shortest_paths)
```

Parameters

Each location $i \in V$ is associated with the following parameters:

- Capacity, c_j: The maximum number of demands that a facility j can handle.
- Demand, d_i: The requirement at location i.

Decision Variables

The following decision variables are defined:

- $x_{ij} \in \{0, 1\}$: Binary variable that indicates if facility j serves location i.
- $y_i \in \{0, 1\}$: Binary variable that indicates if location i is selected as a facility.

Objective Function

The objective is to minimize the total distance covered to serve all locations:

$$\text{Minimize} \quad \sum_{i \in V} \sum_{j \in V} d_{ij} x_{ij}$$

Constraints

The problem is subject to the following constraints:

1. Each demand must be served by exactly one facility:

$$\sum_{j \in V} x_{ij} = 1, \quad \forall i \in V$$

2. A facility must be open to serve demands:

$$x_{ij} \leq y_j, \quad \forall i, j \in V$$

3. Facility capacity constraints:

$$\sum_{i \in V} d_i x_{ij} \leq c_j, \quad \forall j \in V$$

Python Code: Facility Location Model

```
from pyomo.environ import *

# Model definition
model = ConcreteModel()

# Sets and Parameters
model.I = Set(initialize=[1, 2, 3])  # Locations
model.J = Set(initialize=[1, 2])     # Facilities
```

```
model.d = Param(model.I, initialize={1: 5, 2: 10, 3: 7})  # Demands
model.c = Param(model.J, initialize={1: 15, 2: 20})       # Capacities
model.dist = Param(model.I, model.J, initialize={(1, 1): 4, (1, 2): 6,
                                                 (2, 1): 5, (2, 2): 3,
                                                 (3, 1): 8, (3, 2): 2})

# Decision Variables
model.x = Var(model.I, model.J, domain=Binary)
model.y = Var(model.J, domain=Binary)

# Objective Function
def objective_rule(model):
    return sum(model.dist[i, j] * model.x[i, j] for i in model.I for j in model.J)
model.objective = Objective(rule=objective_rule, sense=minimize)

# Constraints
def demand_constraint(model, i):
    return sum(model.x[i, j] for j in model.J) == 1
model.demand_constraint = Constraint(model.I, rule=demand_constraint)

def capacity_constraint(model, j):
    return sum(model.d[i] * model.x[i, j] for i in model.I) <= model.c[j]
model.capacity_constraint = Constraint(model.J, rule=capacity_constraint)

# Solve
SolverFactory('glpk').solve(model)
print("Optimal Objective Value: ", model.objective())
```

Non-Deterministic

Decision Variables

Define:

- x_{ij}: A binary variable that equals 1 if location i is served by facility j, and 0 otherwise.
- y_j: A binary variable that equals 1 if facility j is opened, and 0 otherwise.

Parameters

L is the set of locations.
$D \sim \mathcal{N}(\mu, \sigma^2)$ represent the demand at each location
c_{ij} denote the cost (or distance) of serving location i from facility j.
K is the number of facilities to open.

Objective Function

The objective is to minimize the expected total transportation cost:

$$\text{Minimize } Z = E\left[\sum_{i\in L}\sum_{j\in L} c_{ij}x_{ij}\right]$$

Constraints

1. Demand Satisfaction: Each location's demand must be satisfied by the opened facilities:

$$\sum_{j\in L} x_{ij} = 1, \quad \forall i \in L$$

2. Capacity Constraints: The total demand assigned to any facility cannot exceed its capacity when it is opened:

$$\sum_{i\in L} D_i x_{ij} \leq C_j y_j, \quad \forall j \in L$$

3. Facility Opening Constraint: A location can only serve if its corresponding facility is opened:

$$x_{ij} \leq y_j, \quad \forall i, j$$

4. Number of Facilities: The total number of facilities opened must equal K:

$$\sum_{j} y_j = K$$

Monte Carlo Simulation

To handle the stochastic nature of demand, a Monte Carlo simulation is employed. This involves running multiple iterations where random demands are generated based on a normal distribution.

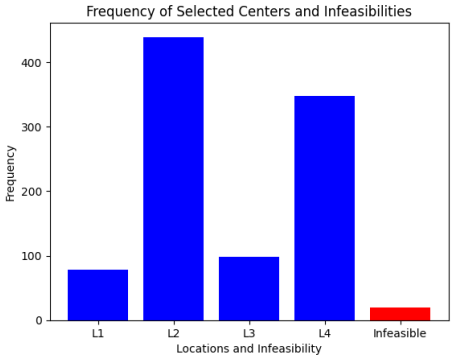
Python code

```
# Define locations and parameters
locations = ['L1', 'L2', 'L3', 'L4']
distances = {
    ('L1', 'L1'): 0, ('L1', 'L2'): 10, ('L1', 'L3'): 20, ('L1', 'L4'): 30,
    ('L2', 'L1'): 10, ('L2', 'L2'): 0, ('L2', 'L3'): 15, ('L2', 'L4'): 25,
    ('L3', 'L1'): 20, ('L3', 'L2'): 15, ('L3', 'L3'): 0, ('L3', 'L4'): 35,
    ('L4', 'L1'): 30, ('L4', 'L2'): 25, ('L4', 'L3'): 35, ('L4', 'L4'): 0
}
capacities = {'L1': 30, 'L2': 30, 'L3': 30, 'L4': 30}
k = 2
mean_demand = 10
```

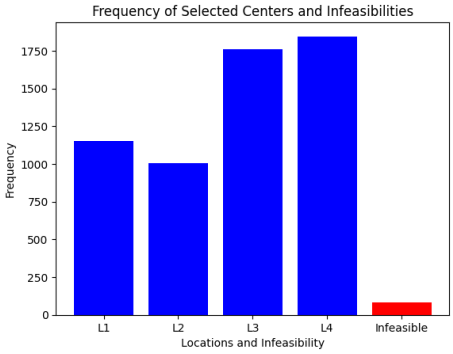
```
std_dev_demand = 5

# Simulation parameters
n_simulations = 500 # Number of Monte Carlo simulations
selected_centers_count = Counter()
infeasible_count = 0

# Monte Carlo simulation
for sim in range(n_simulations):
    # Generate random demands
    demands = {loc: max(0, int(np.random.normal(mean_demand, std_dev_demand))) for loc in lo
    total_demand = sum(demands.values())
    total_capacity = k * max(capacities.values())

    if total_demand > total_capacity:
        infeasible_count += 1
        continue # Skip this iteration if infeasible

    # Create a Pyomo model
    model = ConcreteModel()
    model.Locations = Set(initialize=locations)
    model.Demand = Param(model.Locations, initialize=demands)
    model.Capacity = Param(model.Locations, initialize=capacities)
    model.Distance = Param(model.Locations, model.Locations, initialize=distances)
    model.x = Var(model.Locations, model.Locations, domain=Binary)
    model.y = Var(model.Locations, domain=Binary)

    # Objective
    def objective_rule(model):
        return sum(model.Distance[i, j] * model.x[i, j] for
        i in model.Locations for j in model.Locations)

    model.objective = Objective(rule=objective_rule, sense=minimize)

    # Constraints
    def assignment_rule(model, i):
        return sum(model.x[i, j] for j in model.Locations) == 1

    model.assignment_constraint = Constraint(model.Locations, rule=assignment_rule)

    def capacity_rule(model, j):
        return sum(model.Demand[i] * model.x[i, j] for i in model.Locations) <=
        model.Capacity[j] * model.y[j]

    model.capacity_constraint = Constraint(model.Locations, rule=capacity_rule)
```

```
def open_center_rule(model, i, j):
    return model.x[i, j] <= model.y[j]

model.open_center_constraint = Constraint(model.Locations, model.Locations,
rule=open_center_rule)

def number_of_centers_rule(model):
    return sum(model.y[j] for j in model.Locations) == k

model.num_centers_constraint = Constraint(rule=number_of_centers_rule)

# Solve the model
solver = SolverFactory('glpk')
result = solver.solve(model)

if (result.solver.status == 'ok') and
(result.solver.termination_condition == 'optimal'):
    # Count the selected centers
    for j in model.Locations:
        if model.y[j].value == 1:
            selected_centers_count[j] += 1
```

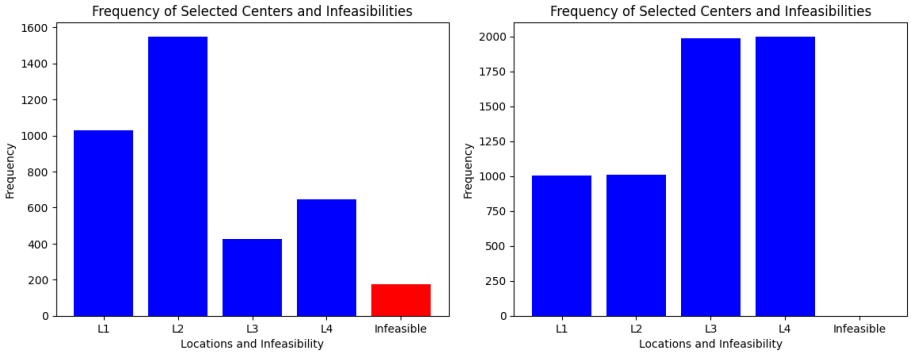
Result



(a) Histogram for $k = 2$, iter $= 500$, demand $N(10, 5)$.



(b) Histogram for $k = 2$, iter $= 2000$, demand $N(18, 2)$, capacity $N(35, 3.5)$.



(a) Histogram for $k = 2$, iter $= 2000$, demand $N(14, 2)$.

(b) Histogram for $k = 2$, iter $= 2000$, demand $N(10, 4)$, capacity $N(30, 3)$.

Figure 2: Comparison of histograms for different configurations of demand and capacity.

Competitive Facility Location Problem

The Competitive Facility Location Problem (CoFLP) focuses on optimizing the placement of facilities by multiple competitors aiming to maximize their market share in a shared region. Each competitor selects a limited number of locations from a set of candidates to serve geographically distributed customer demand, with service determined by proximity within a threshold distance. The problem incorporates constraints on the number of facilities each competitor can open and ensures a minimum number of unique facility placements to reduce overlap and encourage diversity. The goal is to balance competitiveness and efficiency by maximizing the total demand covered while accounting for the interactions between competitors' decisions.

Sets and Indices

- I: Set of candidate facility locations, indexed by i.
- J: Set of customer points, indexed by j.
- K: Set of competitors, indexed by k.

Parameters

- d_{ij}: Distance between facility i and customer j.
- w_j: Demand at customer j.
- λ: Threshold distance within which a facility can cover a customer.
- α_k: Maximum number of facilities competitor k can open.

- min_unique_locations: Minimum number of unique facility locations across all competitors.

Decision Variables

- $x_{k,i} \in \{0,1\}$: Binary variable, $x_{k,i} = 1$ if competitor k places a facility at location i, 0 otherwise.
- $z_{k,i,j} \in \{0,1\}$: Binary variable, $z_{k,i,j} = 1$ if customer j is covered by facility i of competitor k, 0 otherwise.
- $\pi_k \geq 0$: Total demand covered by competitor k.
- $y_i \in \{0,1\}$: Binary variable, $y_i = 1$ if any competitor places a facility at location i, 0 otherwise.

Objective Function

Maximize the total demand covered by all competitors:

$$\max \sum_{k \in K} \pi_k$$

Constraints

1. Coverage Constraint

Each customer j can be covered by at most one facility from a competitor k, provided the facility is within the threshold distance λ:

$$\sum_{i \in I : d_{ij} \leq \lambda} z_{k,i,j} \leq 1, \quad \forall k \in K, \forall j \in J$$

2. Total Demand Coverage

The total demand covered by each competitor k is the sum of the demand of all covered customers:

$$\pi_k = \sum_{j \in J} w_j \cdot \sum_{i \in I : d_{ij} \leq \lambda} z_{k,i,j}, \quad \forall k \in K$$

3. Facility Limit Constraint

Each competitor k can open at most α_k facilities:

$$\sum_{i \in I} x_{k,i} = \alpha_k, \quad \forall k \in K$$

4. Facility Constraint

A customer j can only be covered by a facility i of competitor k if that facility is opened:

$$z_{k,i,j} \leq x_{k,i}, \quad \forall k \in K, \forall i \in I, \forall j \in J$$

5. Linking Facilities to Unique Locations

If any competitor opens a facility at location i, the corresponding y_i variable must be set to 1:

$$y_i \geq \frac{1}{|K|} \sum_{k \in K} x_{k,i}, \quad \forall i \in I$$

6. Minimum Unique Locations Constraint

The total number of unique facility locations used across all competitors must be at least min_unique_locations:

$$\sum_{i \in I} y_i \geq \text{min\_unique\_locations}$$

Variable Domains

$$\begin{aligned} x_{k,i} &\in \{0,1\}, \quad \forall k \in K, \forall i \in I \\ z_{k,i,j} &\in \{0,1\}, \quad \forall k \in K, \forall i \in I, \forall j \in J \\ y_i &\in \{0,1\}, \quad \forall i \in I \\ \pi_k &\geq 0, \quad \forall k \in K \end{aligned}$$

Python Code

```
from pyomo.environ import *
from pyomo.opt import SolverFactory

# Problem Parameters
n_customers = 10  # Number of customer points
n_facilities = 10  # Number of candidate facility locations
m_competitors = 2  # Number of competitors
alpha = {1: 2, 2: 2}  # Number of facilities each competitor can open
lambda_threshold = 2  # Maximum distance for coverage
min_unique_locations = 3  # Minimum unique facility locations across competitors
customer_demand = {j: 1 for j in range(1, n_customers + 1)}  # Demand at each point

# Distance matrix (random values here; replace with actual distances)
import random
random.seed(42)
distances = {(i, j): random.randint(1, 5) for i in
```

```
range(1, n_facilities + 1) for j in range(1, n_customers + 1)}

# Set of candidate locations and customer points
I = range(1, n_facilities + 1)
J = range(1, n_customers + 1)

# Pyomo Model
model = ConcreteModel()

# Sets
model.I = Set(initialize=I)  # Facility locations
model.J = Set(initialize=J)  # Customer points
model.K = Set(initialize=range(1, m_competitors + 1))  # Competitors

# Parameters
model.distances = Param(model.I, model.J, initialize=distances)
model.customer_demand = Param(model.J, initialize=customer_demand)
model.lambda_threshold = lambda_threshold
model.alpha = Param(model.K, initialize=alpha)

# Variables
model.x = Var(model.K, model.I, domain=Binary)  # Whether competitor k places
a facility at i
model.z = Var(model.K, model.I, model.J, domain=Binary)  # Whether customer j is covered
by competitor k at facility i
model.pi = Var(model.K, domain=NonNegativeReals)  # Total demand covered by each
competitor
model.y = Var(model.I, domain=Binary)  # Whether any facility is opened at location i

# Objective: Maximize total demand coverage for both competitors
model.objective = Objective(expr=sum(model.pi[k] for k in model.K), sense=maximize)

# Constraints
def coverage_constraint(model, k, j):
    """Ensure customer demand is covered by at most one facility per competitor."""
    return sum(model.z[k, i, j] for i in model.I if model.distances[i, j] <=
    model.lambda_threshold) <= 1

model.coverage_constraint = Constraint(model.K, model.J, rule=coverage_constraint)

def demand_coverage_constraint(model, k):
    """Calculate total demand covered by each competitor."""
    return model.pi[k] == sum(
        model.customer_demand[j] *
        sum(model.z[k, i, j] for i in model.I if model.distances[i, j] <=
        model.lambda_threshold) for j in model.J)
```

```
model.demand_coverage_constraint = Constraint(model.K, rule=demand_coverage_constraint)

def facility_limit_constraint(model, k):
    """Limit the number of facilities opened by each competitor."""
    return sum(model.x[k, i] for i in model.I) == model.alpha[k]

model.facility_limit_constraint = Constraint(model.K, rule=facility_limit_constraint)

def facility_activation_constraint(model, k, i, j):
    """Customer can be covered only if a facility exists at the location."""
    return model.z[k, i, j] <= model.x[k, i]

model.facility_activation_constraint = Constraint(model.K, model.I, model.J,
rule=facility_activation_constraint)

#Constraints for Unique Locations
def link_y_to_x_constraint(model, i):
    """Link y[i] to competitors' facility decisions."""
    return model.y[i] >= sum(model.x[k, i] for k in model.K) / m_competitors

model.link_y_to_x_constraint = Constraint(model.I, rule=link_y_to_x_constraint)

def min_unique_locations_constraint(model):
    """Ensure a minimum number of unique facility locations."""
    return sum(model.y[i] for i in model.I) >= min_unique_locations

model.min_unique_locations_constraint = Constraint(rule=min_unique_locations_constraint)

# Solver and Results
solver = SolverFactory('glpk')  # Use an appropriate solver
results = solver.solve(model, tee=True)

# Display Results
print("\n Optimization Results ")

# Print total demand covered by each competitor
for k in model.K:
    print(f"Competitor {k}:")
    print(f"  Total Demand Covered: {model.pi[k].value:.2f}")
    print(f"  Facilities Opened at:", [i for i in model.I if model.x[k, i].value == 1])
print()

# Facility-location matrix
print("Facility-Location Matrix (x[k, i]):")
location_matrix = [[model.x[k, i].value for i in model.I] for k in model.K]
```

```
for k in model.K:
    print(f"Competitor {k}: {location_matrix[k - 1]}")
print()

# Print unique locations
unique_locations = [i for i in model.I if model.y[i].value > 0.5]
print(f"Unique Facility Locations Used: {unique_locations}")
```

Result

For the above set parameters the result is:

```
Optimization Results
Competitor 1:
  Total Demand Covered: 9.00
  Facilities Opened at: [2, 6]
Competitor 2:
  Total Demand Covered: 9.00
  Facilities Opened at: [1, 10]

Facility-Location Matrix (x[k, i]):
Competitor 1: [0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
Competitor 2: [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]

Unique Facility Locations Used: [1, 2, 6, 10]
```

For the parameters:

```
n_customers = 10
n_facilities = 10
m_competitors = 3
alpha = {1: 2, 2: 1, 3:2}
lambda_threshold = 2
min_unique_locations = 3
customer_demand = {j: 1 for j in range(1, n_customers + 1)}
```

The result is:

```
Optimization Results
Competitor 1:
  Total Demand Covered: 7.00
  Facilities Opened at: [2, 10]
Competitor 2:
  Total Demand Covered: 7.00
  Facilities Opened at: [1]
Competitor 3:
  Total Demand Covered: 9.00
  Facilities Opened at: [4, 9]
```

```
Facility-Location Matrix (x[k, i]):
Competitor 1: [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0]
Competitor 2: [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
Competitor 3: [0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]
```