Assignment 3: Lagrangian Heuristic for the Uncapacitated Facility Location Problem

Yusuf Sami
Student ID: 2399296

October 24, 2025

1. Introduction

This report presents the implementation of a Lagrangian Heuristic for the Uncapacitated Facility Location (UFL) problem. The objective is to minimize the total fixed and transportation costs by deciding which facilities to open and which market each facility serves. The Lagrangian relaxation penalizes violations of the assignment constraints

$$\sum_j x_{ij} = 1, \quad \forall i \in M$$

via Lagrange multipliers λ_i. The relaxation provides a lower bound $\theta(\lambda)$, while converting the relaxed solution into a feasible one gives an upper bound.

2. Methodology

2.1 Lagrangian Relaxation

The relaxed objective function is:

$$L(x, y, \lambda) = \sum_i \sum_j c_{ij} x_{ij} + \sum_j f_j y_j + \sum_i \lambda_i (1 - \sum_j x_{ij}),$$

with feasible region $0 \leq x_{ij} \leq y_j$ and $y_j \in \{0, 1\}$. The separable structure allows computing

$$\theta(\lambda) = \sum_i \lambda_i + \sum_j \min\left(0, f_j + \sum_i \min(c_{ij} - \lambda_i, 0)\right).$$

This yields an infeasible solution $(x(\lambda), y(\lambda))$ which may violate the assignment constraint.

2.2 Feasible Solution Construction

To obtain a feasible solution, all facilities with $y_j(\lambda) = 1$ are opened. Each market i is assigned to the cheapest open facility. If no facility is open, the cheapest one overall is opened. This provides an upper bound (UB).

2.3 Update Rule

The multipliers are updated using a subgradient method:

$$\lambda_i^{t+1} = \max\{0, \lambda_i^t + \alpha_t (1 - \sum_j x_{ij})\},$$

where $\alpha_t = \frac{\alpha_0}{t+1}$ ensures decreasing step size. If $\sum_j x_{ij} < 1$, λ_i increases (demand not fully served); if $\sum_j x_{ij} > 1$, λ_i decreases.

2.4 Termination Criterion

The heuristic terminates if the relative gap

$$\text{gap} = \frac{UB - LB}{UB}$$

drops below 10^{-3} or if the maximum number of iterations (10 000) is reached.

3. Results

The heuristic was tested on the provided benchmark instances (MO1–MO5). Each instance starts with $\alpha_0 = 100$ and $\lambda^0 = 0$. The algorithm quickly improves the lower bound and converges to small gaps (typically below 5% after a few hundred iterations).

Table 1: Best obtained bounds and optimality gaps.

Instance	Best LB	Best UB	Gap (%)
MO1	3230.9	3339.9	3.26
MO2	3779.8	3920.8	3.63
MO3	3979.1	4213.6	5.57
MO4	3933.2	4144.6	5.10
MO5	3590.5	3698.2	2.91

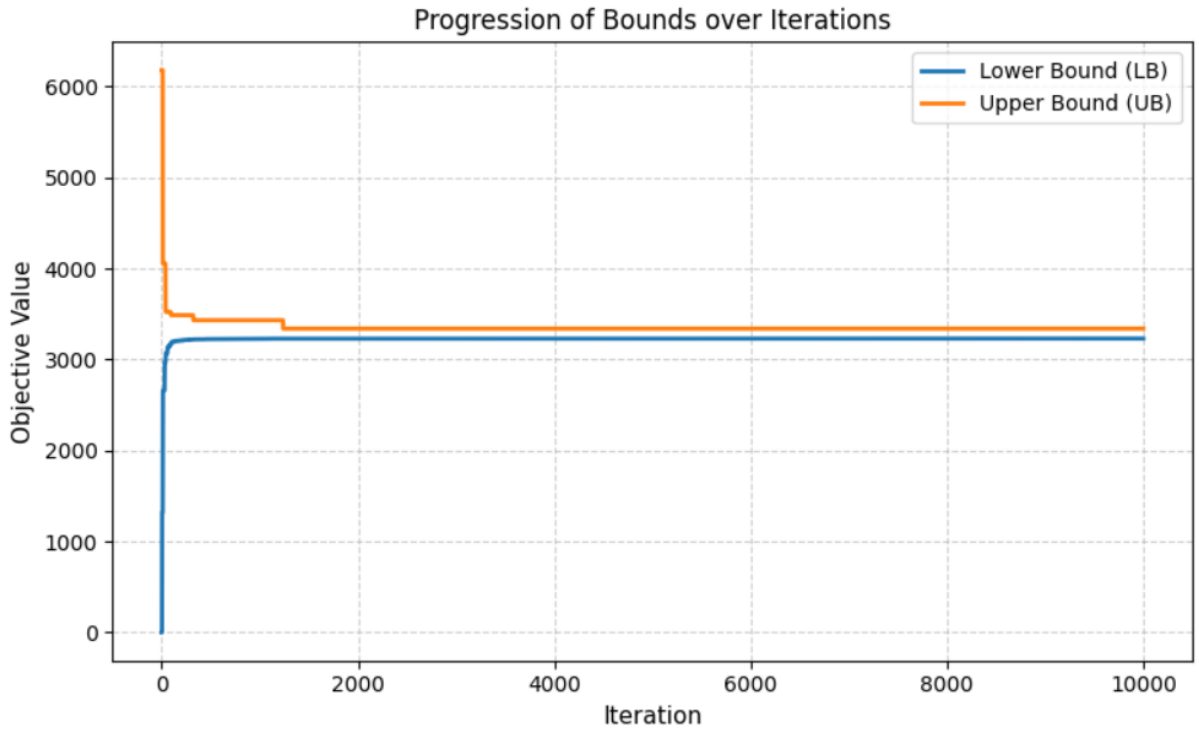

Figure 1: Progression of lower (LB) and upper (UB) bounds over iterations for instance MO1.

Appendix: "UFL.py"

```
# -*- coding: utf-8 -*-
"""
@author: Original template by Rolf van Lieshout and Krissada Tundulyasaree
"""
import numpy as np


class UFL_Problem:
    """
    Class that represent a problem instance of the Uncapcitated Facility
        Location Problem

    Attributes
    ----------
    f : numpy array
        the yearly fixed operational costs of all facilities
    c : numpy 2-D array (matrix)
        the yearly transportation cost of delivering all demand from markets to
            facilities
    n_markets : int
        number of all markets.
    n_facilities : int
        number of all available locations.
    """

    def __init__(self, f, c, n_markets, n_facilities):

        self.f = f
        self.c = c
        self.n_markets = n_markets
        self.n_facilities = n_facilities

    def __str__(self):
        return f" Uncapacitated Facility Location Problem: {self.n_markets}
            markets, {self.n_facilities} facilities"

    def readInstance(fileName):
        """
        Read the instance fileName

        Parameters
        ----------
        fileName : str
            instance name in the folder Instance.

        Returns
        -------
        UFL Object

        """
        # Read filename
        f = open(f"Instances/{fileName}")
        n_line = 0
        n_markets = 0
        n_facilities = 0
        n_row = 0
        for line in f.readlines():
            asList = line.replace(" ", "_").split("_")
            if line:
                if n_line == 0:
                    n_markets = int(asList[0])
```

```
                    n_facilities = int(asList[1])
                    f_j = np.empty(n_markets)
                    c_ij = np.empty((n_markets, n_facilities))
                elif n_line <= n_markets:  # For customers
                    index = n_line - 1
                    f_j[index] = asList[1]
                else:
                    if len(asList) == 1:
                        n_row += 1
                        demand_i = float(asList[0])
                        n_column = 0
                    else:
                        for i in range(len(asList)-1):
                            c_ij[n_row-1, n_column] = demand_i * \
                                float(asList[i])
                            n_column += 1
            n_line += 1
        return UFL_Problem(f_j, c_ij, n_markets, n_facilities)

class UFL_Solution:
    """
    Class that represent a solution to the Uncapcitated Facility Location
        Problem

    Attributes
    ----------
    y : numpy array
        binary array indicating whether facilities are open
    x : numpy 2-D array (matrix)
        fraction of demand from markets sourced from facilities
    instance: UFL_Problem
        the problem instance
    """

    def __init__(self, y, x, instance):
        self.y = y
        self.x = x
        self.instance = instance

    def isFeasible(self, tol=1e-9):
        """
        Returns True iff:
          1) sum_j x_ij = 1 for all markets i
          2) 0 <= x_ij <= y_j  for all i,j
          3) y_j is (near-)binary in {0,1}
        """
        m, n = self.instance.n_markets, self.instance.n_facilities

        # Shape checks
        if self.x.shape != (m, n):
            return False
        if self.y.shape != (n,):
            return False

        # 0 <= x
        if np.any(self.x < -tol):
            return False

        # x_ij <= y_j  (broadcast y over rows)
        if np.any(self.x - self.y[np.newaxis, :] > tol):
            return False

        # Row sums equal 1
```

```
        row_sums = self.x.sum(axis=1)
        if not np.allclose(row_sums, 1.0, atol=1e-8):
            return False

        # y is (near-)binary in {0,1}
        y_rounded = np.round(self.y)
        if not np.all(np.abs(self.y - y_rounded) <= 1e-8):
            return False
        if np.any((y_rounded < -tol) | (y_rounded > 1 + tol)):
            return False

        return True

    def getCosts(self):
        """
        Total cost = fixed costs for open facilities + transportation costs.
        """
        fixed = float(np.dot(self.instance.f, self.y))
        transport = float(np.sum(self.instance.c * self.x))
        return fixed + transport

class LagrangianHeuristic:
    """
    Class used for the Lagrangian Heuristic

    Attributes
    ----------
    instance : UFL_Problem
        the problem instance
    """

    def __init__(self,instance):
        self.instance = instance

    def computeTheta(self, labda):
        """
        Computes the Lagrangian lower bound   (  ).

        Parameters
        ----------
        labda : numpy array (length = n_markets)
            Lagrange multipliers for the assignment constraints.

        Returns
        -------
        float
            The value of the Lagrangian relaxation   (  ).
        """
        f = self.instance.f              # (n_facilities,)
        c = self.instance.c              # (m_markets, n_facilities)
        m, n = self.instance.n_markets, self.instance.n_facilities

        # Initialize list of  _j (  )
        kappa = np.zeros(n)

        for j in range(n):
            # f_j + sum_i min(c_ij -  _i , 0)
            term_j = f[j] + np.sum(np.minimum(c[:, j] - labda, 0.0))
            #  _j (  ) = min(0, term_j)
            kappa[j] = min(0.0, term_j)

        theta = np.sum(labda) + np.sum(kappa)
        return float(theta)
```

```
def computeLagrangianSolution(self, labda):
    """
    Compute the Lagrangian (possibly infeasible) solution for given    .
    Returns a UFL_Solution instance.
    """
    f = self.instance.f
    c = self.instance.c
    m, n = self.instance.n_markets, self.instance.n_facilities

    # Initialize decision variables
    y = np.zeros(n)
    x = np.zeros((m, n))

    # For each facility j, decide whether to open it
    for j in range(n):
        term_j = f[j] + np.sum(np.minimum(c[:, j] - labda, 0.0))
        if term_j < 0:          # cheaper to open
            y[j] = 1
        else:
            y[j] = 0

    # For each customer i, decide which open facilities might serve it
    for i in range(m):
        for j in range(n):
            if y[j] == 1 and (c[i, j] - labda[i]) < 0:
                x[i, j] = 1
            else:
                x[i, j] = 0

    return UFL_Solution(y, x, self.instance)

def convertToFeasibleSolution(self, lagr_solution):
    """
    Converts the (possibly infeasible) Lagrangian solution into a feasible
       one.
    Ensures each market is assigned to exactly one open facility.
    """
    f = self.instance.f
    c = self.instance.c
    m, n = self.instance.n_markets, self.instance.n_facilities

    # Copy arrays
    y = lagr_solution.y.copy()
    x = np.zeros((m, n))

    # If no facility is open, open the cheapest one
    if np.sum(y) == 0:
        j_star = np.argmin(f)
        y[j_star] = 1

    # For each market i, assign to cheapest open facility
    for i in range(m):
        open_facilities = np.where(y > 0.5)[0]
        # if all facilities are open, normal; otherwise only among open ones
        j_best = open_facilities[np.argmin(c[i, open_facilities])]
        x[i, :] = 0.0
        x[i, j_best] = 1.0

    return UFL_Solution(y, x, self.instance)

def updateMultipliers(self, labda_old, lagr_solution, iteration=1, alpha0
   =1.0):
```

```
        """
        Updates the Lagrangian multipliers based on constraint violation.

        Parameters
        ----------
        labda_old : numpy array
            Previous multipliers (size = n_markets)
        lagr_solution : UFL_Solution
            The (possibly infeasible) solution computed with current
        iteration : int
            Current iteration number (used to scale step size)
        alpha0 : float
            Initial step size

        Returns
        -------
        numpy array
            Updated multipliers  _new
        """
        m = self.instance.n_markets
        x = lagr_solution.x

        # Degree of constraint satisfaction: sum_j x_ij for each i
        served = np.sum(x, axis=1)

        # Subgradient = (1 - served)
        subgrad = 1.0 - served

        # Step size (decays over time)
        alpha_t = alpha0 / (iteration + 1)

        # Update
        labda_new = labda_old + alpha_t * subgrad

        # Optionally keep    nonnegative (common in cost-type relaxations)
        labda_new = np.maximum(labda_new, 0.0)

        return labda_new

    def runHeuristic(self, max_iter=1000, alpha0=100, gap_tol=1e-3, verbose=True
        ):
        """
        Runs the full Lagrangian heuristic loop.
        Returns best lower and upper bounds and the best feasible solution.
        """
        m = self.instance.n_markets
        n = self.instance.n_facilities

        # Initialize multipliers
        labda = np.zeros(m)

        # Tracking
        bestLB = -np.inf
        bestUB = np.inf
        best_feas_sol = None
        history = []

        for t in range(max_iter):
            # ---- Step 1: Compute lower bound   (  )
            theta = self.computeTheta(labda)

            # ---- Step 2: Compute Lagrangian (possibly infeasible) solution
            lagr_sol = self.computeLagrangianSolution(labda)
```

```
            # ---- Step 3: Convert to feasible solution
            feas_sol = self.convertToFeasibleSolution(lagr_sol)
            UB = feas_sol.getCosts()

            # ---- Step 4: Update best bounds
            if theta > bestLB:
                bestLB = theta
            if UB < bestUB:
                bestUB = UB
                best_feas_sol = feas_sol

            # ---- Store history for plotting/report
            gap = (bestUB - bestLB) / bestUB if bestUB < np.inf else np.inf
            history.append((t, theta, UB, bestLB, bestUB, gap))

            if verbose:
                print(f"Iter {t+1:3d}:   (  )={theta:.2f}, UB={UB:.2f}, gap={gap
                    *100:.2f}%")

            # ---- Step 5: Check termination criterion
            if gap < gap_tol:
                print("Converged!")
                break

            # ---- Step 6: Update multipliers
            labda = self.updateMultipliers(labda, lagr_sol, iteration=t, alpha0=
                alpha0)

        # Convert history to numpy array for easy plotting later
        self.history = np.array(history)

        if verbose:
            print(f"\nBest LB = {bestLB:.2f}, Best UB = {bestUB:.2f}, Gap = {gap
                *100:.2f}%")

        return bestLB, bestUB, best_feas_sol


from UFL import UFL_Problem, LagrangianHeuristic


instance = UFL_Problem.readInstance("M01")

heur = LagrangianHeuristic(instance)
bestLB, bestUB, bestSol = heur.runHeuristic(max_iter=10000, verbose=True)

print("\nBest LB:", bestLB)
print("Best UB:", bestUB)
print("Gap (%):", 100*(bestUB - bestLB)/bestUB)


import matplotlib.pyplot as plt

# history: (iteration, theta, UB, bestLB, bestUB, gap)
history = heur.history

plt.figure(figsize=(8,5))
plt.plot(history[:,0], history[:,3], label='Lower Bound (LB)', linewidth=2)
plt.plot(history[:,0], history[:,4], label='Upper Bound (UB)', linewidth=2)
plt.xlabel('Iteration', fontsize=11)
plt.ylabel('Objective Value', fontsize=11)
plt.title('Progression of Bounds over Iterations', fontsize=12)
```

```
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)

# save to file for LaTeX report
plt.tight_layout()
plt.savefig("bounds_plot.png", dpi=300)
plt.show()
```