MA4254 Discrete Optimization Computational Assignment

John Lee

November 2023

Consider the Facility Location Problem modelled as a MILP and the alternative and equivalent MILP formulation known as the Aggregate Facility Location Problem.

Facility Location Problem (FLP)

$$
\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} & \\
\text{subject to:} & \displaystyle\sum_{j=1}^{n} x_{i,j} = 1 & \forall i \\
& x_{i,j} \leqslant y_j & \forall i, j \\
& 0 \leqslant x_{i,j} \leqslant 1, \quad y_j \in \{0,1\} &
\end{array}
$$

Aggregate Facility Location Problem (AFL)

$$
\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} & \\
\text{subject to:} & \displaystyle\sum_{j=1}^{n} x_{i,j} = 1 & \forall i \\
& \displaystyle\sum_{i=1}^{m} x_{i,j} \leqslant m y_j & \forall j \\
& 0 \leqslant x_{i,j} \leqslant 1, \quad y_j \in \{0,1\} &
\end{array}
$$

(a) The number of linear inequalities in the FLP is $mn + 2$ since $x_{i,j} \leqslant y_j$ gives mn inequalities and $0 \leqslant x_{i,j} \leqslant 1$ gives 2 extra inequalities. The number of linear inequalities in the AFL is $n + 2$. For $m > 1$, the number of inequalities in FLP is greater than the number of inequalities in AFL, and equal when $m = 1$.

(b) To show that the two sets of feasible solutions for FLP and AFL are equivalent, let us look at the extended form of the inequality constraints for both.

FLP:

$$
\begin{gathered}
x_{1,1} \leqslant y_1 \\
x_{2,1} \leqslant y_1 \\
\vdots \\
x_{m,1} \leqslant y_1 \\
\vdots \\
x_{m,n} \leqslant y_n
\end{gathered}
$$

AFL:

$$
\begin{gathered}
x_{1,1} + x_{2,1} + \ldots + x_{m,1} \leqslant m y_1 \\
x_{1,2} + x_{2,2} + \ldots + x_{m,2} \leqslant m y_2 \\
\vdots \\
x_{1,m} + x_{2,m} + \ldots + x_{m,n} \leqslant m y_n
\end{gathered}
$$

To see these two inequality constraints are equivalent, simply sum the first m FLP inequalities, this is equivalent to the 1st inequality constraint for AFL. Then sum the second m FLP inequalities, this is equivalent to the 2nd inequality constraint for AFL, and so on. Therefore, the two sets of feasible solutions for FLP and AFL are equivalent.

(c), (d) By replacing the binary constraints $y_j \in \{0, 1\}$ with the box constraint $0 \leqslant y_i \leqslant 1$ we derive the linear relaxation to FLP and AFL:
FLP-LR:

$$
\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} & \\
\text{subject to:} & \displaystyle\sum_{j=1}^{n} x_{i,j} = 1 & \forall i \\
& x_{i,j} \leqslant y_j & \forall i, j \\
& 0 \leqslant x_{i,j} \leqslant 1, \quad 0 \leqslant y_j \leqslant 1 &
\end{array}
$$

AFL-LR:

$$
\begin{aligned}
&\text{Minimize:} && \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} && \\
&\text{subject to:} && \sum_{j=1}^{n} x_{i,j} = 1 && \forall i \\
& && \sum_{i=1}^{m} x_{i,j} \leqslant m y_j && \forall j \\
& && 0 \leqslant x_{i,j} \leqslant 1, \quad 0 \leqslant y_j \leqslant 1 &&
\end{aligned}
$$

(e) Arranging the optimal values of (FLP-Val), (AFL-Val), (FLP-LR-Val), and (AFL-LR-Val) in increasing order we get (AFL-LR-Val)$\leqslant$(FLP-LR-Val)$\leqslant$(FLP-Val)=(AFL-Val)

(f) For this question, we use Python's numpy library to generate random facility locations and random customer locations, by using `np.random.rand(m)`, and `np.random.rand(n)`, respectively, where `m=25`, `n=15`. Then we initialise a distance matrix with all zeros and used two for loops to compute distances and fill the distance matrix. View file for code for the distance matrix. Fig 1 is a display of the generated data.

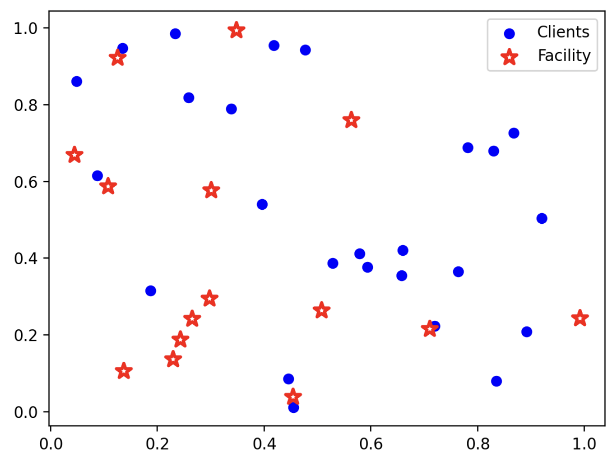

Fig 1

(g), (h) Fig. 2 is a visualization of the solution to the FLP. Fig. 3 shows a table which compares FLP-Val to FLP-LR-Val. The code uses the Gurobi optimization library (`gurobipy`) to solve a facility location problem. We start by creating a `run_simulation` function, which generates random locations for customers and facilities, along with associated setup costs (fixed to equal 1), and shipping costs equal to the euclidean distance between two points. We then create a Mixed Integer Linear Programming (MILP) model to solve the facility location problem. The MILP model is defined with binary decision variables for facility selection (`select`) and continuous decision variables for assignment (`assign`). Constraints are added to ensure that each customer is assigned to exactly one facility and that the assignment variables are bounded. The MILP model is then optimized using `m_MIP.optimize()`. The relaxation of the MILP model is created using `m_LP = m_MIP.relax()` and then optimized using `m_LP.optimize()`. The code then initializes lists (`mip_results` and `lp_results`) to store the MILP and LP relaxation objective values for each simulation, and a counter (`same_results_count`) to keep track of how many times the MILP and relaxation results are the same. We create an empty DataFrame (`results_df`) to store the MILP and relaxation results. The code runs the simulation 100 times in a loop and it checks if the MILP and relaxation results are close (within a small tolerance) and increments `same_results_count` if they are, and found that they are the same roughly 99 times out of 100.

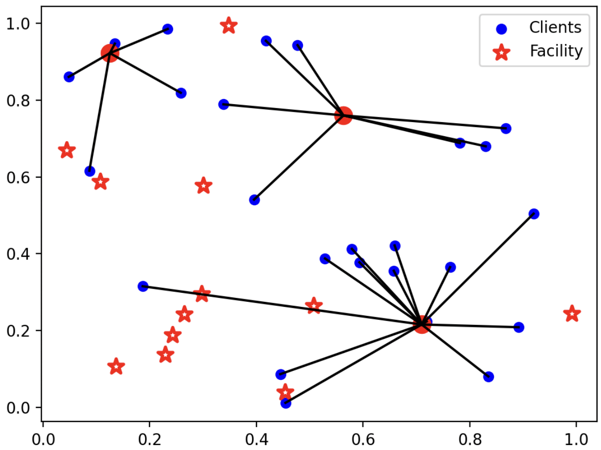

Fig 2

```
    MIP Objective  LP Relaxation Objective
0       20.077562                20.077562
1       17.816056                17.816056
2       15.736706                15.736706
3       15.443645                15.443645
4       17.765598                17.765598
..            ...                      ...
95      16.720072                16.720072
96      17.380737                17.380737
97      18.623450                18.623450
98      18.871468                18.871468
99      17.329025                17.329025

[100 rows x 2 columns]
Average MIP Objective: 17.48376273942474
Average LP Relaxation Objective: 17.483727199313986
Number of times results are the same: 99
```

Fig 3

For the AFL, since we have proved earlier it is equivalent to the FLP, we simply used the same code to solve the AFL. For the relaxation of AFL, we simply change the inequality constraints and ran the code 100 times and generated this table.

85	8.65921
86	8.18267
87	9.2258
88	8.2125
89	8.84761
90	8.13072
91	8.06745
92	7.45477
93	8.48314
94	7.92263
95	8.73246
96	8.01431
97	8.4555
98	8.15871
99	8.1343
100	7.95102
Average	8.54927

Fig 4

In Fig.4 I have only shown a part of the full table. View code for the full table of 100 optimum values. As we can see the AFL-LR has much lower optimum values than the AFL and different opt. values in all 100 runs.

(i) The Capacitated Facility Location Problem (CFLP) has MILP formulation

Capacitated Facility Location Problem (CFLP)

$$
\begin{aligned}
&\text{Minimize:} && \sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} && \\
&\text{subject to:} && \sum_{j=1}^{n} x_{i,j} = 1 && \forall i \\
& && \sum_{i=1}^{m} x_{i,j} \leqslant r_j y_j && \forall i, j \\
& && 0 \leqslant x_{i,j} \leqslant 1, \quad y_j \in \{0,1\}, \quad r_j \in \mathbb{Z}_{\geqslant 0} &&
\end{aligned}
$$

Relaxed Capacitated Facility Location Problem (CFLP-LR)

$$
\begin{array}{lll}
\text{Minimize:} & \displaystyle\sum_{j=1}^{n} c_j y_j + \sum_{i,j=1}^{n,m} d_{i,j} x_{i,j} & \\
\text{subject to:} & \displaystyle\sum_{j=1}^{n} x_{i,j} = 1 & \forall i \\
 & \displaystyle\sum_{i=1}^{m} x_{i,j} \leqslant r_j y_j & \forall i, j \\
 & 0 \leqslant x_{i,j} \leqslant 1, \quad r_j \in \mathbb{Z}_{\geqslant 0} &
\end{array}
$$

(j) To implement the code to solve CFLP and relaxed CFLP we use the PuLP library for linear programming. Then we set `r = 2` which specifies that each facility has a capacity of 2. Then use `model = pulp.LpProblem("Facility_Location_Problem", pulp.LpMinimize)` which creates an empty linear programming problem named `"Facility_Location_Problem"` to minimize. `x` and `y` are defined as decision variables representing binary variables indicating whether a client is assigned to a facility or whether a facility is opened, respectively. The rest follows a similar structure as FLP with added constraint `model += pulp.lpSum(x[i, j] for i in range(m)) <= r * y[j]`.

```
      Run  Optimal Value (MILP)  Optimal Value (Relaxed LP)
0     1.0            331.823969                  331.823969
1     2.0            328.548407                  328.548407
2     3.0            328.739124                  328.739124
3     4.0            328.980953                  328.980953
4     5.0            328.940407                  328.940407
..    ...                   ...                         ...
95   96.0            330.243327                  330.243327
96   97.0            329.114839                  329.114839
97   98.0            328.923810                  328.923810
98   99.0            330.142879                  330.142879
99  100.0            329.095989                  329.095989

[100 rows x 3 columns]
Number of times solutions are the same: 100
```

Fig 4

Travelling Salesman Problem

(a) We begin by creating a function that takes as input a tour and outputs the total distance. We start by using `import random` which provides functions for generating random numbers. Then, `generate_distance_matrix(n)`: This is a function that takes an integer `n` as input and returns a 2D list (matrix) representing distances between `n` points. The distances are randomly generated. We start by initialising the matrix to be the zero matrix, and fill in entries using two nested loops iterate over the rows and columns of the matrix, using randomly generated integers between 1 and 20 (just used an as example).

`calculate_total_distance(tour, distance_matrix):` is a function which calculates the total distance of a given tour based on the provided distance matrix. `total_distance = 0` initializes the total distance variable. `n = len(tour)` gets the length of the tour, which represents the number of points to visit. `tour[i]-1` and `tour[(i+1)%n]-1` are used to index the `distance_matrix`. The -1 is used because the tour points are 1-based, but Python uses 0-based indexing. `tour[(i+1)%n]` ensures that the last point in the tour connects back to the first point. Then, for sake of simple checking, I used

```
tour = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

as an example tour and checked my output was the same as the sum of the diagonal entries 1 above the main diagonal and the entry $a_{10,1}$.

(b) Next, we create the function `generate_perturbed_tour(tour):` this is the first perturbation method (in the code file I also created a function for the second perturbation method). It takes a list called `tour` as input and returns a perturbed tour. `n = len(tour)` gets the length of the input tour. `i = random.randint(0, n - 2)` randomly selects an index `i` such that $0 \leqslant i \leqslant n-2$. This will be used to split the tour. `j = random.randint(i + 1, n - 1)` randomly selects an index j such that $i < j \leqslant n-1$. This ensures that j is after i in the tour.

The perturbed tour is constructed as:

```
perturbed_tour = tour[:i] + tour[i:j+1][::-1] + tour[j+1:]
```

Here,

- `tour[:i]` takes the elements before i as they are.
- `tour[i:j+1][::-1]` reverses the sub-tour between i and j.
- `tour[j+1:]` takes the elements after j as they are.

Finally, the function returns the `perturbed_tour`.

(c)(d) See code file with comments in code. To generate random cities, we construct the function `def generate_random_cities(n):` which returns `return [(random.uniform(0, 1), random.uniform(0, 1)) for _ in range(n)]`. When we implement the code, we set `n=50`. After some experimentation, we use parameters:

- `max_iterations = 10000`
- `initial_temperature = 10000`
- `cooling_rate = 0.995`

and get on average `Total_distance` between 5-7.

(e) Then we plot the solution using `import matplotlib.pyplot as plt` and creating function `def plot_solution(cities, solution):` which first plots the cities as red dots and then loops through the `solution` which is a list representing the order of visiting cities. `plt.plot(..., 'k-')` specifies that the lines should be drawn in a solid black line. Fig. 5 shows the results.

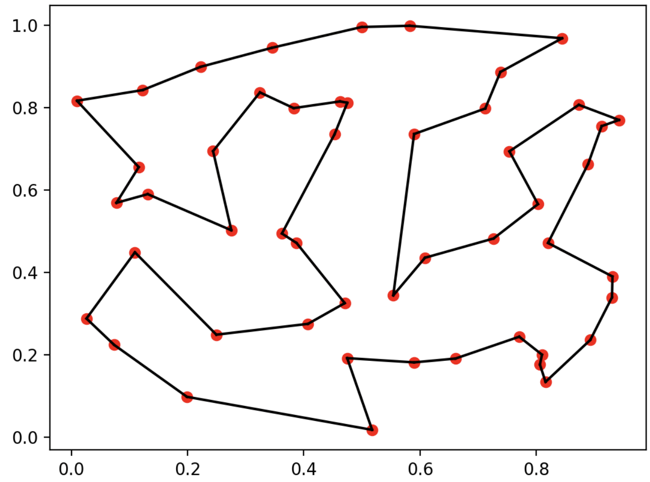

Fig. 5

With this generation of cities, we see there are no self intersection of edges.

(f) Next we modify the perturbation function. We define a function named `generate_perturbed_tour` which takes one argument: `input_tour` and is similar to our original `generate_perturbed_tour` except for the line

```
perturbed_tour = input_tour[:i] + input_tour[j:j+1][::-1] + input_tour[i+1:j]
                 + input_tour[i:i+1] + input_tour[j+1:]
```

By replacing `generate_perturbed_tour` with `generate_perturbed_tour_2` in our Simulated Annealing algorithm we get on average `Total_distance` between 8-10 with multiple self intersections as can be seen in fig. 6.

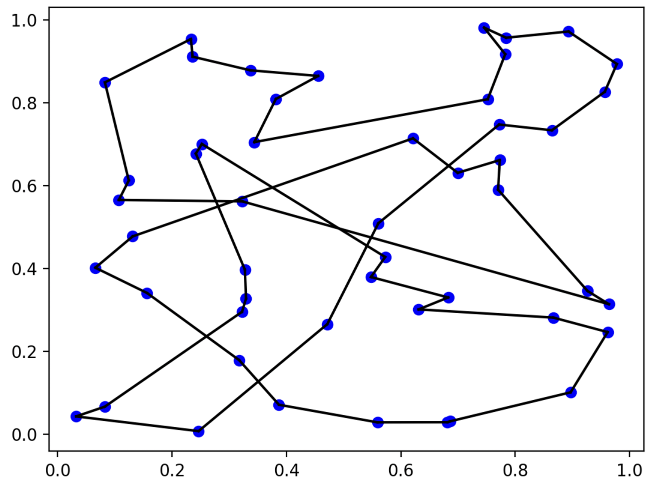

Fig. 6

Citations

- Lecture 3: Linear Programming relaxations and rounding. (n.d.). `https://www.cs.dartmouth.edu/~deepc/Courses/F10/Notes/lec3.pdf`

- Google. (n.d.). Google colaboratory. Google Colab. https://colab.research.google.com/drive/1YWJ8MT8t9-9bOOZKEkrchMfu0YLZ4zJv?usp=sharing#scrollTo=eaXSAbu1TTwi
- Facility location problem. Facility location problem - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.). https://optimization.cbe.cornell.edu/index.php?title=Facility_location_problem
- Documentation. Gurobi Optimization. (2023, October 4). https://www.gurobi.com/documentation/
- Pulp documentation. Pulp Documentation - Pulp Project 3.40.1 documentation. (n.d.). https://docs.pulpproject.org/pulpcore/
- Brownlee, J. (2021, October 11). Simulated annealing from scratch in Python. MachineLearningMastery.com. https://machinelearningmastery.com/simulated-annealing-from-scratch-in-python/
- James, N. (2022, April 20). Simulated annealing algorithm explained from scratch (python) - Machine Learning Plus. https://www.machinelearningplus.com/machine-learning/simulated-annealing-algorithm
- Google. (n.d.-a). Google colaboratory. Google Colab. https://colab.research.google.com/drive/1LPZ_0T7AuTPWSL4RpqQhzHGD3HC7isVX?usp=sharing