FM216 Final Project Report

The p-Median Facility Allocation Problem

Dhruv Srivastava (U20220033)

Pranjal Rastogi (U20220086)

Satvik Bajpai (U20220103)

Tanay Srinivasa (U20220086)

Declaration

We, Dhruv Srivastava, Pranjal Rastogi, Satvik Bajpai and Tanay Srinivasa, certify that this project is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this project has not previously been submitted for assessment in any academic capacity, and that we have not copied in part or whole or otherwise plagiarised the work of other persons. We confirm that we have identified and declared all possible conflicts that we may have.

Signed and submitted

Abstract

In this paper, we explore the p-median facility location problem. Location planning is a complex problem that requires governments and planning agencies to consider multiple factors simultaneously. In this project, we aim to offer an optimization-based solution for facility location planning. The question we address is: How can we effectively balance the costs of setting up facilities and transportation to achieve optimal facility locations? If we imagine the world to be a graph, then the problem thus becomes one of finding the p-median-vertices of a weighted graph. Hence, called the p-median problem. Instead of stopping at finding the apt locations for a fixed number of medians, we also present our unique take on the problem where we try to find out the ideal number of medians and where exactly they should be located. We explored two variants of this problem: ”The Uncapacitated Problem” and ”The Capacitated Problem.” In the uncapacitated problem, we assume medians to have infinite capacity; that is, they can supply infinite demand. The idea behind the capacitated problem is that a facility supplying a higher demand must have a higher facility cost. Hence, we define the facility cost as a function of the demand fulfilled, the production per unit area, and the cost per unit area of the facility.

Solving these problems with direct enumeration turns computationally intense for large values of n, as you would have to calculate and compare the costs of n choose p combinations of medians. As n increases, these combinations increase exponentially. Hence, we use two heuristics to solve the problem: the Greedy Heuristic and the Vertex Substitution Heuristic. We compare the speed and accuracy of these heuristics with enumeration to understand the use cases of each heuristic. To illustrate the real-world application of this problem, we demonstrate the selection of distribution centres for Nandini Milk Parlours in the Bangalore Sales Depot area. We found the distance between these nodes and assigned the facility cost of each node based on the average registry cost of the area in which the node is located. We compare the output of the different heuristics, comparing their accuracy.

Keywords

p-median, Warehouse Location, Distribution, Supply and Demand, Heuristic, Spatial Distribution

Acknowledgements

We extend our gratitude to Viraj D. Souza for providing valuable feedback and guidance during the iterative updates of this work, and providing us with the initial idea to work on this problem. Special thanks to Dr. Nitin Upadhyaya for offering invaluable insights and support as course instructor.

Moreover, we acknowledge the contributions of Teitz and Bart, as well as Keuhn, whose papers on heuristics significantly informed and enriched this project.

Contents

Declaration **i**

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**

1.1 Introduction to Location Planning 1

1.2 The Uncapacitated Problem 2

1.2.1 Description 2

1.2.2 Cost Function 3

1.3 The Capacitated Problem 4

1.3.1 Description 4

1.3.2 Cost Function 5

1.4 The Need for Heuristics 5

2 Solving The Uncapacitated Problem **6**

2.1 Greedy Heuristic 6

2.1.1 Pseudo-code 6

2.1.2 Solved Example 7

2.2 Vertex Substitution Heuristic 8

2.2.1 Pseudo-code 8

2.2.2 Solved Example 10

2.3 Enumeration 13

2.3.1 Pseudo-code 13

2.3.2 Solved Example 14

3 Solving The Capacitated Problem **14**

3.1 Greedy-Like Heuristic 15

3.1.1 Pseudo-code 15

3.1.2 Solved Example 16

3.2 Other Heuristics for Solving this Variant 19

4 Comparison of Heuristics **19**
4.1 Runtime 19
4.2 Accuracy 22

5 Real World Application **23**
5.1 Problem Definition 23
5.2 Code Output 25
5.3 Inference of Output and Conclusions 26

6 Conclusion **26**
6.1 Future Scope 27

References **27**

Appendix A: Supplemental Materials **29**

List of Figures

1 Measuring runtime while varying p from 2 to 15 while keeping $N = 15$ 20
2 Measuring runtime while varying N from 3 to 20 while keeping $p = 5$. 21
3 Measuring runtime while varying N from 3 to 20 while keeping $p = 17$ 21
4 Measuring runtime while varying N from 3 to 49 while keeping $p = 3$, for only the heuristics 22
5 Standardized residuals for the heuristics when $N = 10$ and $p = 5$ 23
6 Real World Application Definition 25
7 Solutions to the Real World Problem (Selected Heuristics circled in Yellow) 26

1 Introduction

In the following manuscript, we tackle a problem known as the ”p-median allocation problem problem”. Mathematically, this problem deals with the idea of finding p-medians of a weighted graph. In practice, this can be applied to the context of Location Planning, which is explored here.

The code associated with this project is included in Appendix A. It is also maintained at https://github.com/PjrCodes/pmedian_opt. Instructions on how to use the code is available in the repository’s `README.md`.

1.1 Introduction to Location Planning

Location planning is a complex problem that requires governments and planning agencies to look at multiple factors at the same time. In this project, we aim to provide an optimisation-based solution for the location planning of facilities. A facility is any system that provides a service, such as fuel or healthcare. Costs of setting up facilities and transportation of products play a large role in planning out the locations of facilities in an area. Cost of setting up differs from location to location, and so does transportation costs. This fundamental issue can be framed as a discrete optimisation problem known as the p-median problem.

The p-median problem is a distance-based optimisation problem related to Graph Theory in which p nodes need to be located on a graph of N nodes in such a way that they minimize the costs at the graph vertices, and the costs (distances) along the graph edges.

In our problem, the N nodes are nothing but N demand points (customers) who need to be serviced by p facilities. The p facilities need to be assigned to the customers in such a way that each customer is connected to a single facility, and the costs are minimized. When demand is taken into consideration, the cost is the sum of the demand-weighted distance (demand * distance) between all customers and corresponding facilities. Further, the cost of setting up a facility (or, the cost of running a facility for an year) can be regarded as the cost at the graph vertex. When p facilities are assigned, they are located at certain demand points. Our aim is to find these locations and the number of such facilities. That is, what number (and which) of the p medians give the least cost.

To solve the p-median problem, we can manipulate several variables as assumptions. These include: the cost of setting up facilities (yearly cost), the demand of customers, production per unit area, and more. We have identified two variants of the problem based on the manipulation of these assumptions.

The first variant is a ”simple” version of the problem, in which the customer demands are assumed to be *equal*, and only cost of setting up a facility (yearly) is considered. This variant is called The Uncapacitated Problem (Section 1.2). In the second, ”complex” variant, customer demand is considered *unequal* and facility capacity is considered via production per unit area and cost per unity area. This variant is termed as The Capacitated Problem (Section 1.3). The variants are explained in further detail in the subsequent sections.

1.2 The Uncapacitated Problem

1.2.1 Description

The uncapactitated problem involves determining the optimal selection of p facilities from a given set, considering **equal demand** from each customer and **infinite capacity** for each facility (hence the name). The objective is to minimise the total cost of a ”configuration”, where a configuration is a set of p selected locations in a system of N possible locations (demand points). This is done using the distance matrix (akin to an adjacency matrix in Graph Theory) and cost array (for setting-up costs) provided, while ensuring that each customer is allocated to only one facility and each facility caters to the same type of demand (commodity).

Inputs:

- Number of Nodes (Demand Points) - an integer N.
- Distance Matrix between Nodes i to j - a symmetric matrix of real numbers D of size N. i and j index over N.
- Cost per year of setting up at Median j - an array C of real numbers of size N. j indexes over N.

Outputs:

- Selected Medians - An array of p indexes, where the index indicates which node is selected.
- Total Cost - A real number indicating the total cost of the configuration.

Constraints:

- Each customer has equal demand.
- Each customer can fulfil their demand from only one facility.
- Each facility fulfils the same kind of demand or commodity.
- Capacity of each facility is assumed infinite (there is no restriction)

1.2.2 Cost Function

The cost function that is minimized in this problem can be described as follows,

$$\text{Cost}(X, D, Y, C) = \sum_{j=1}^{N} \sum_{i=1}^{N} x_{i,j} d_{i,j} + \sum_{j=1}^{N} y_j c_j \tag{1}$$

Here,

$$X_{N\times N} : \text{where } x_{ij} \in X = \begin{cases} 1, & \text{if the node } i \text{ is fulfilled by median } j \\ 0, & \text{otherwise} \end{cases}$$

$$D_{N\times N} : \text{where } d_{ij} \in D = \text{ distance from node } i \text{ to node } j$$

$$Y_N : \text{where } y_j \in Y = \begin{cases} 1, & \text{if node } j \text{ is a chosen median} \\ 0, & \text{otherwise} \end{cases}$$

$$C_N : \text{where } c_j \in C = \text{yearly cost of setting up median } j$$

1.3 The Capacitated Problem

1.3.1 Description

This variant of the p-median problem involves determining the optimal selection of p facilities (medians) from a given set of nodes, considering their respective production per unit area, (yearly) cost per unit area, and the distance matrix between these facilities and customers (adjacency matrix). The goal is to satisfy the varying demands of customers while minimizing the total cost incurred. The idea is to select facilities in those locations where costs for production per unit area are minimum and the demands are satisfied evenly with the least transportation cost (distance).

Inputs:

- Number of Nodes (Demand Points/ Customers) - an integer N
- Distance Matrix between Nodes i to j - a symmetric matrix of real numbers D of size N. i and j index over N.
- Cost per unit area for Median j - an array C of real numbers of size N. j indexes over N.
- Production per unit area for Median j - a array P of real numbers of size N. j indexes over N.
- Demand of every customer node i - a array W of real numbers of size N. i indexes over N.

Outputs:

- Selected Medians - An array of p indexes, where the index indicates which node is selected.
- Total Cost - A real number indicating the total cost of the configuration.

Constraints:

- Each facility has a fixed, different production per unit area.
- Each facility has a fixed, different cost per unit area.
- Each customer has unequal demand.

- Each facility fulfils the same kind of demand or commodity.

1.3.2 Cost Function

The cost function that is minimized in this problem can be described as follows,

$$\text{Cost}(X, D, Y, C, A) = \sum_{j=1}^{N} \sum_{i=1}^{N} x_{ij} d_{ij} + \sum_{j=1}^{N} y_j a_j c_j \tag{2}$$

Here,

$$\begin{aligned}
X_{N\times N} : \text{where } x_{ij} \in X &= \begin{cases} 1, & \text{if the node } i \text{ is fulfilled by median } j \\ 0, & \text{otherwise} \end{cases} \\
D_{N\times N} : \text{where } d_{ij} \in D &= \text{Distance from node } i \text{ to node } j \\
Y_N : \text{where } y_j \in Y &= \begin{cases} 1, & \text{if node } j \text{ is a chosen median} \\ 0, & \text{otherwise} \end{cases} \\
C_N : \text{where } c_j \in C &= \text{Cost per unit area of setting up median } j \\
T_N : \text{where } t_j \in T_N &= \text{Total demand fulfilled by median } j \\
P_N : \text{where } p_j \in P &= \text{Production per unit area of median } j \\
A_N : \text{where } a_j \in A &= \frac{t_j}{p_j}
\end{aligned}$$

1.4 The Need for Heuristics

To solve the above problems, naïve thinking can lead us to the brute-force approach of ”direct enumeration.” In this, we go through every possible subset of p medians. This is a computationally intense system, in which $\binom{n}{p}$ combinations will have to be checked. Hence, heuristic algorithms have been developed and explored to optimally solve the problems.

We have used two heuristics in our comparisions. A brief about the heuristics used is given below.

1. **The Greedy Heuristic** [5]:

   Idea: Starts with no facilities and adds destinations one at a time, choosing the one leading to the maximum decrease in the cost function.

2. **Vertex Substitution Heuristic** [9]:

   Idea: Facilities in the solution are exchanged with those not in the solution, evaluating the solution after each interchange.

2 Solving The Uncapacitated Problem

Let's solve the uncapacitated problem using the heuristics detailed above.

2.1 Greedy Heuristic

This heuristic is a modified version of the one presented by Kuehn and Hamburger in [5].

2.1.1 Pseudo-code

Greedy Cost:

```
INPUT k = Selected Medians
LET Cost = 0
LET V_SEEN = []
FOR EACH m IN k:
    Z_i = { 0,       if the node i is in V_SEEN
          { D_{m,i}, otherwise
    INCREMENT Cost BY \sum_i^N Z_i + c_m
    ADD k TO V_SEEN
RETURN Cost
```

$$Z_i = \begin{cases} 0, & \text{if the node i is in V\_SEEN} \\ D_{m,i}, & \text{otherwise} \end{cases}$$

Greedy Solver:

```
INPUT R = All Nodes
LET SELECTED = []
FOR EACH value FROM 1 TO p:
    M = INDEX(MIN(Greedy Cost(R)))
    ADD M TO SELECTED
    REMOVE M FROM R
RETURN SELECTED
```

Compute Cost:

X = matrix such that $x_{ij} = \begin{cases} 1, & \text{if } j \text{ in SELECTED and } D_{ij} < D_{i(k!=j)} \\ 0, & \text{otherwise} \end{cases}$

RETURN $\sum_i^N \sum_j^N x_{ij} d_{ij} + \sum_j y_j c_j$

2.1.2 Solved Example

Let's use the following values for the example,

$$N = 3$$
$$D = \begin{bmatrix} 0 & 5 & 10 \\ 5 & 0 & 7 \\ 10 & 7 & 0 \end{bmatrix}$$
$$C = \begin{pmatrix} 5 & 7 & 5 \end{pmatrix}$$

For $p = 1$:

$$\text{SELECTED} = \text{index}\left(\min \begin{bmatrix} 5+10+5 \\ 5+7+7 \\ 10+7+5 \end{bmatrix}\right) = 2$$
$$\text{Configuration Cost} = 19$$

For $p = 2$:

$$\begin{aligned}
\text{SELECTED}_1 &= 2 \\
\text{SELECTED}_2 &= \min \begin{bmatrix} 10+5 \\ 10+5 \end{bmatrix} = \text{Median } j = 1 \\
\therefore\ \text{SELECTED} &= \begin{bmatrix} 2 & 1 \end{bmatrix} \\
\text{Cost} &= 19
\end{aligned}$$

For $p = 3$:

$$\begin{aligned}
\text{SELECTED} &= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
\text{Cost} &= 17
\end{aligned}$$

Hence, the selected medians are **[1, 2, 3]** and cost is 17.

2.2 Vertex Substitution Heuristic

This heuristic is presented by Teitz and Bart in [9].

2.2.1 Pseudo-code

N = number of nodes
c_i = recurring cost of running the facility every year
V = All Nodes
For each pair of nodes i, j the minimum path distance d_{ij} is found. These d_{ij}'s are used to create the distance matrix D.
A weight matrix H is made with diagonal entries as c_i and non-diagonal entries as 0.
R is the total cost matrix and is given by H $\times$ D.

Now, V_1 = subset of p nodes from V
For each vertex v_j in V_1, an associated set of vertices P_{1j} is found such that:

$$P_{1j} = \{v_i | r_{ij} < r_{ik} \forall v_k \in V_1\}$$

This basically finds the vertices in V which are closest to v_j. P_1 is thus a partition of V.

Compute the cost of this partition using:

$$\sum_{v_j \in V_1} \sum_{v_i \in P_1} r_{ij}$$

Then, construct the sub-matrix of R, "R_p" by adjoining the p columns corresponding to V_1.

The source from which any destination v_i is served can be defined as v_k such that:

$$r_{ik} \leq r_{ij} \forall v_k, v_j \in V_p$$

This is the same as the i^th^ row minimum in the sub-matrix R_p. The total weighted distance r_m for the m^th^ source subset will be the sum of these row minima.

Now, replace one vertex v_j in the source subset V_1 by another v_b.
If r_{ij} were not the i^th^ row minimum of R_p then no change in the i^th^ row contribution to r would result.

However, if r_{ij} were the i^th^ row minimum of R_p then its replacement by r_{ib} might have several outcomes depending on whether:

1. $r_{ib} \leq r_{ij}$
2. $r_{ij} \leq r_{ib} \leq r_{is}$
3. $r_{ij} \leq r_{is} \leq r_{ib}$

where r_{is} is the second smallest i^th^ row element in R_p.

Case 1 the i^th^ row contribution to r from the substitution of v_b from v_j is now incremented by ${}_i\Delta_{bj} = r_{ij} - r_{ib}$ and ${}_i\Delta_{bj} \leq 0$

Case 2 the i^th^ row contribution to r from the substitution of v_b from v_j is now incremented by ${}_i\Delta_{bj} = r_{ij} - r_{ib}$ and ${}_i\Delta_{bj} > 0$

Case 3 the i^th^ row contribution to r from the substitution of v_b from v_j is now incremented by ${}_i\Delta_{bj} = r_{ij} - r_{is}$ and ${}_i\Delta_{bj} > 0$. Here, r_{is} becomes the i$^{\text{th}}$ row minimum.

Whether it is worth substituting v_b instead of v_j depends upon the net effect of the increments summed over all rows:

$$\Delta_{bj} = \sum_i \Delta_{bj} \text{ (Note: this is for one } v_j\text{)}$$

Substituting v_b for v_j is only conducted when works only if $\Delta_{bj} < 0$

In the Vertex Substitution Heuristic, this "cycle" is repeated to ensure the least cost has been reached based on the assumption. Therefore, the pseudocode can be written as,

1. **LET** V_1 = random initial source vertex subset of size p
2. **FOR EACH** $v_j \in V_1$:
   COMPUTE P_{1j}, and add to P_1
3. **COMPUTE** cost r_1 for the partitioned system P_1
4. **SELECT** $v_b \notin V_1$
5. **FOR EACH** $v_j \in V_1$, substitute v_b and **COMPUTE** Δ_{bj}
6. **SELECT** v_k **FROM** V_1 **SUCH THAT**:
   $\Delta_{bk} < 0$ and $\Delta_{bk} = \min \Delta_{bj}$ where $j = \{1, 2, 3, \ldots p\}$
7. **IF** v_k can be selected, **THEN SUBSTITUTE** v_k **BY** v_b in V_1.
   LET this be V_2. Compute the cost r_2
   ELSE don't change V_1
8. **SELECT** another vertex $v_b \notin V_1, V_2$ and not previously tried (some vertex might have been tried and then rejected altogether).
9. **REPEAT** step (5) to (8) **UNTIL** no more new vertices can be tried. The resulting set is defined as the final V_1.
10. With final V_1, **REPEAT** steps (2) to (9). This is a *cycle*.
11. **STOP** when a cycle results in no change in cost r.

RETURN V_1

2.2.2 Solved Example

Taking the same values as used in Section 2.1.2, we get the following outputs,

```
For p=2
==================== Cycle 1 ====================
V1 = [0 1]
partitions=[[0], [1, 2]]
Submatrix Rp = [[ 0. 35.]
 [25.  0.]
 [50. 49.]]

Trying to substitute 2 inside current set [0 1].

Trying 0 -> 2. [2 1]
For row i=0, ith delta = 35.0
For row i=1, ith delta = 0
For row i=2, ith delta = 0
Delta for configuration = 35.0

Trying 1 -> 2. [0 2]
For row i=0, ith delta = 0
For row i=1, ith delta = 25.0
For row i=2, ith delta = -49.0
Delta for configuration = -24.0

Found a good replacement, 1 -> 2
VS Cost = 35.0
==================== Cycle 2 ====================
V1 = [0 2]
partitions=[[0, 1], [2]]
Submatrix Rp = [[ 0. 50.]
 [25. 35.]
 [50.  0.]]

Trying to substitute 1 inside current set [0 2].
```

```
Trying 0 -> 1. [1 2]
For row i=0, ith delta = 35.0
For row i=1, ith delta = -25.0
For row i=2, ith delta = 0
Delta for configuration = 10.0

Trying 2 -> 1. [0 1]
For row i=0, ith delta = 0
For row i=1, ith delta = 0
For row i=2, ith delta = 49.0
Delta for configuration = 49.0

Did not find any good replacements.
VS Cost = 25.0
==================== Cycle 3 ====================
V1 = [0 2]
partitions=[[0, 1], [2]]
Submatrix Rp = [[ 0. 50.]
 [25. 35.]
 [50.  0.]]

Trying to substitute 1 inside current set [0 2].

Trying 0 -> 1. [1 2]
For row i=0, ith delta = 35.0
For row i=1, ith delta = -25.0
For row i=2, ith delta = 0
Delta for configuration = 10.0

Trying 2 -> 1. [0 1]
For row i=0, ith delta = 0
For row i=1, ith delta = 0
For row i=2, ith delta = 49.0
```

```
Delta for configuration = 49.0

Did not find any good replacements.

VS Cost = 25.0
Cost is same as last cycle, ending computation.

For p=3
==================== Cycle 1 ====================
V1 = [0 1 2]
partitions=[[0], [1], [2]]
VS Cost = 0.0
==================== Cycle 2 ====================
V1 = [0 1 2]
partitions=[[0], [1], [2]]
VS Cost = 0.0
Cost is same as last cycle, ending computation.

cost=15 selected_medians=array([0, 2]) p=2 no_ops=3
```

Since the above is zero-indexed, the selected medians are **[1, 3]** and cost is **15**. It is important to note here that the Vertex Substitution heuristic **does not work** for $p = 1$.

2.3 Enumeration

For creating a baseline, we also solved the problem using the brute-force enumeration method, as detailed below.

2.3.1 Pseudo-code

Enumerate Solver

```
INPUT R = All Nodes
LET SELECTED = []
```

```
LET BestCost = ∞
FOR k IN (R choose p):
    IF BestCost > ComputeCost(k) THEN
        SELECTED = k
        BestCost = CostFunction(P)
RETURN SELECTED, BestCost
```

Here, `CostFunction` is the same function as defined for the Greedy Solver in Section 2.1.1.

2.3.2 Solved Example

We can solve the problem defined in Section 2.1.2 as follows:

Cost of different combinations of the Nodes:

Choosing Node 1 Cost = (5 + 10 + 5) = 20

Choosing Node 2 Cost = (5 + 7 + 7) = 19

Choosing Node 3 Cost = (10 + 7 + 5) = 22

Choosing Nodes (1, 2) Cost = (5 + 7) + (7) = 19

Choosing Nodes (2, 3) Cost = (7 + 5) + (5) = 17

Choosing Nodes (1, 3) Cost = (5 + 5) + (5) = 15

Choosing Nodes (1, 2, 3) Cost = (5 + 7 + 5) + (0) = 17

Thus, the selected medians are **[1, 3]** and their cost is **15**. This is the same as Vertex Substitution. It appears that the heuristic has given a highly accurate answer. The comparision between heuristics is further explored in Section 4.

3 Solving The Capacitated Problem

We implemented a variation of the Greedy Heuristic to solve this variant.

3.1 Greedy-Like Heuristic

The "Greedy-Like" Heuristic presented here is a variation on the Greedy Heuristic so that it incorporates the complexities of this problem in cost calculation.

3.1.1 Pseudo-code

Greedy Cost

```
INPUT k = selected medians
LET Cost = 0
LET V_SEEN = []

FOR EACH m IN k:
```

$$Z_i = \begin{cases} 0, & \texttt{if node i is in V\_SEEN} \\ D_{m,i}, & \texttt{otherwise} \end{cases}$$

$$E_i = \begin{cases} W_i, & \texttt{if node i NOT in V\_SEEN} \\ 0, & \texttt{otherwise} \end{cases}$$

INCREMENT Cost BY $\sum_i Z_i + \frac{(\sum_i E_i) P_m}{C_m}$

```
    ADD k TO V_SEEN

RETURN Cost
```

Greedy-Like Solver

```
INPUT R = All Nodes
LET SELECTED = []

FOR EACH value FROM 1 TO p:
    LET M = INDEX(MIN(Greedy-Like Cost(R)))
    ADD M TO SELECTED
    REMOVE M FROM R

RETURN SELECTED
```

Compute Cost

$$WW_{ij} = \frac{W_i | P_j}{C_j}$$

$WD_{ij} = WW_{ij} + D_{ij}$

$X_{ij} = \begin{cases} \texttt{1,} & \texttt{if j in SELECTED and } WD_{ij} < WD_{i(k!=j)} \\ \texttt{0,} & \texttt{otherwise} \end{cases}$

$DD_j = W_i \cdot X_{ij} + W_j$

$A_j = \frac{DD_j}{P_j}$

RETURN $\sum_i \sum_j x_{ij} d_{ij} + \sum_j y_j a_j c_j$

3.1.2 Solved Example

We define the example problem for this variant in the following manner:

$$D = \begin{bmatrix} 0 & 5 & 10 \\ 5 & 0 & 7 \\ 10 & 7 & 0 \end{bmatrix}$$
$$C = \begin{pmatrix} 5 & 7 & 5 \end{pmatrix}$$
$$W = \begin{pmatrix} 4 & 6 & 5 \end{pmatrix}$$
$$P = \begin{pmatrix} 1.5 & 2 & 1.7 \end{pmatrix}$$

Then, for $p = 1$,

$$Z = D_{ij}$$

$$E = \begin{bmatrix} 4 & 6 & 5 \\ 4 & 6 & 5 \\ 4 & 6 & 5 \end{bmatrix}$$

$$\text{SELECTED} = \text{index}\left(\min \begin{bmatrix} 5 + 10 + (\frac{4+6+5}{1.5} \times 1) \\ 5 + 7 + (\frac{4+6+5}{2} \times 2) \\ 10 + 7 + (\frac{4+6+5}{1.7} \times 1) \end{bmatrix}\right) = \begin{pmatrix} 1 \end{pmatrix}$$

$$\text{Cost} = 25$$

And for $p = 2$,

$$Z = \begin{bmatrix} 0 & 5 & 10 \\ 0 & 0 & 7 \\ 0 & 7 & 0 \end{bmatrix}$$

$$E = \begin{bmatrix} 0 & 6 & 5 \\ 0 & 6 & 5 \\ 0 & 6 & 5 \end{bmatrix}$$

$$\text{SELECTED} = \text{index}\left(\min\begin{bmatrix} 5+7+(\frac{6+5}{2}\times 2)) \\ 10+7+(\frac{6+5}{1.7}\times 1) \end{bmatrix}\right) = \begin{pmatrix} 1 & 3 \end{pmatrix}$$

$$WW_{ij} = \begin{bmatrix} 2.6 & 4 & 2.35 \\ 4 & 6 & 3.5 \\ 3.3 & 5 & 2.9 \end{bmatrix}$$

$$WD_{ij} = \begin{bmatrix} 2.6 & 9 & 12.35 \\ 9 & 6 & 10.5 \\ 13.3 & 12 & 2.9 \end{bmatrix}$$

$$X_{ij} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$Y_j = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}$$

$$\text{Cost} = 5 + \left(\frac{10}{1.5}\times 1\right) + \left(\frac{5}{1.7}\times 1\right) = 14.607$$

And finally, for $p = 3$,

$$\begin{aligned}
\text{SELECTED} &= \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \\
X_{ij} &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
Y_j &= \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \\
\text{Cost} &= \left(\frac{6}{2} \times 2\right) + \left(\frac{10}{1.5} \times 1\right) + \left(\frac{5}{1.7} \times 1\right) = 11.607
\end{aligned}$$

Hence the selected medians are **[1,2,3]**.

3.2 Other Heuristics for Solving this Variant

While exploring pre-existing literature, a similar heuristic was found in [5]. Though the Vertex Substitution heuristic can be modified for use with this fixed capacity approach, no such algorithm was found, and we plan to apply techniques similar to above on the Vertex Substitution heuristic to see if a ”Vertex Substitution-Like” heuristic can be created for this capacitated variant.

4 Comparison of Heuristics

Two factors stand out when trying to compare algorithms: *runtime* and *accuracy*. A summary of our comparisions between the above heuristics and the brute-force approach on these metrics is provided below.

4.1 Runtime

Our problem has two major inputs that determine runtime: p, the number of medians to select and N, the number of total nodes.

Fixing $N = 15$ but varyin p, we get the graph in Figure 1.

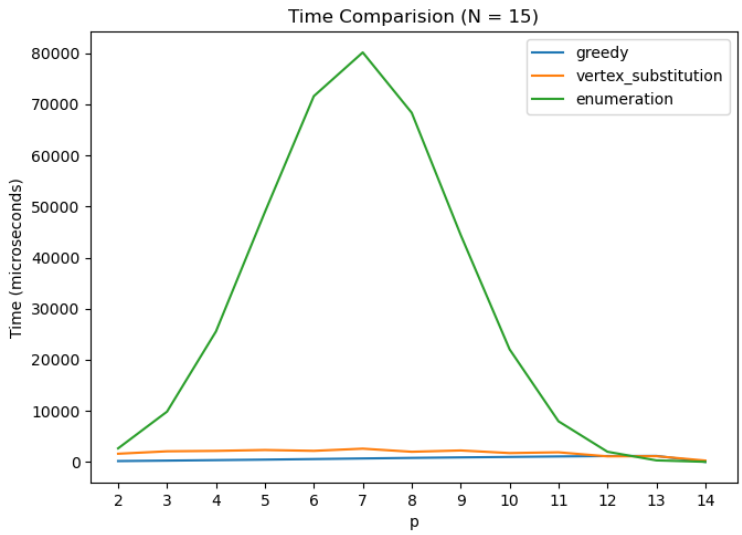


Figure 1: Measuring runtime while varying p from 2 to 15 while keeping $N = 15$

From the graph in Figure 1, it is clear that the enumeration heuristic takes a very high amount of time when $p \approx \frac{N}{2}$. This can be attributed to the fact that $\binom{N}{k}$ is maximum when $k = \frac{N}{2}$. The vertex substitution heurisic seems to be slower than the greedy heuristic, but both of them have a neglgible time difference and seem to be almost linear in runtime complexity.

Now, let's fix p but vary N. We will plot two graphs here, one for $p = 5$ and one for $p = 17$.

In Figure 2, we can clearly see the fact that the brute-force enumeration solver increases exponentially with an increase in N, while both vertex substitution and greedy increase almost linearly. Vertex substitution starts taking a significantly higher amount of time as compared to the greedy heuristic for very large values of N.

Figure 3 is similar to the earlier graph in Figure 2, but from both of these graphs we can notice that the heuristic solvers do not give an answer for cases when $p > N$. Obviously, this is a case in which we can't solve the p-median problem, since we can't choose, say, 7 medians from a set of 5. Hence, these graphs are only really useful after $N = p$ on the x-axis.

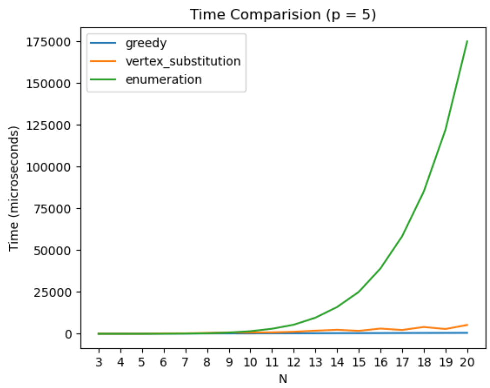


Figure 2: Measuring runtime while varying N from 3 to 20 while keeping $p = 5$

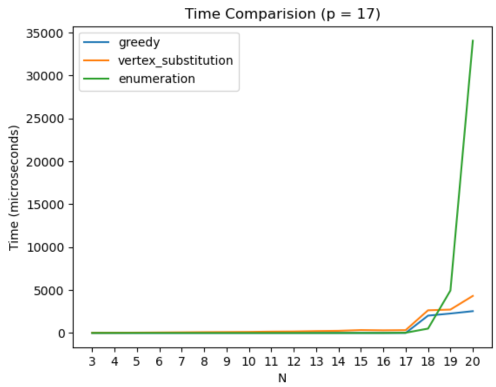


Figure 3: Measuring runtime while varying N from 3 to 20 while keeping $p = 17$

To look into the differences between the heuristics in further detail, we can also zoom into the above graph for only the heuristics. The graph in Figure 4 is obtained when we vary N from 3 to 49, and keep $p = 3$.

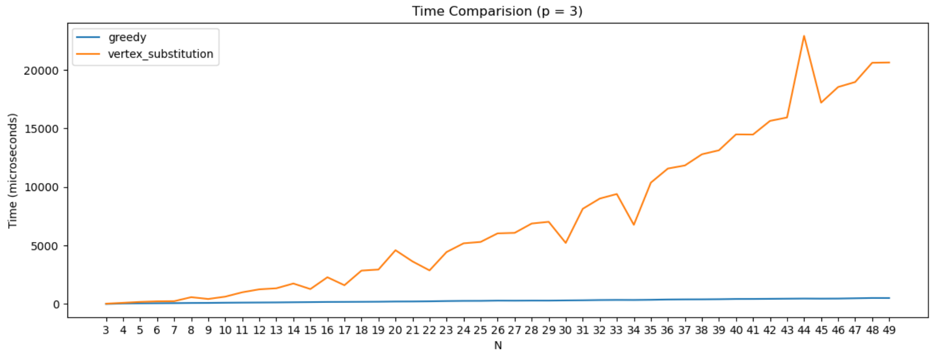


Figure 4: Measuring runtime while varying N from 3 to 49 while keeping $p = 3$, for only the heuristics

Using Figure 4, it can be approximated that the Vertex Substitution heuristic has a linear relationship with N, while the Greedy algorithm seems to remain constant with increases in N! This makes the Greedy Heuristic highly efficient for large values of N, while Vertex Substitution is more thorough and hence takes a longer amount of time.

4.2 Accuracy

Theoretically, it can be seen that Enumeration (brute-force) is the *most accurate*, while the Greedy heuristic is the least accurate. The Vertex Substitution heurisic falls in-between. This idea was visible in our earlier solved example, where the vertex substitution heuristic gave a more accurate answer as compared to the Greedy heuristic (See Sections 2.1.2, 2.2.2 and 2.3.2).

The brute-force enumeration solver's answer can be used as a baseline for comparing accuracies, since it goes over every combination. However, due to the immense time complexity associated with the enumeration solver, it is extremely hard to create large test-cases for this comparision.

Running for 1000 randomized inputs and plotting the standardized residuals, the plot in Figure 5 does not show any clear association or trends in the residuals. Further, it appears that the "accuracy" of Greedy and Vertex Substitution is similar, barring some outliers. More analysis on this problem and the heuristics is required for a thorough understanding

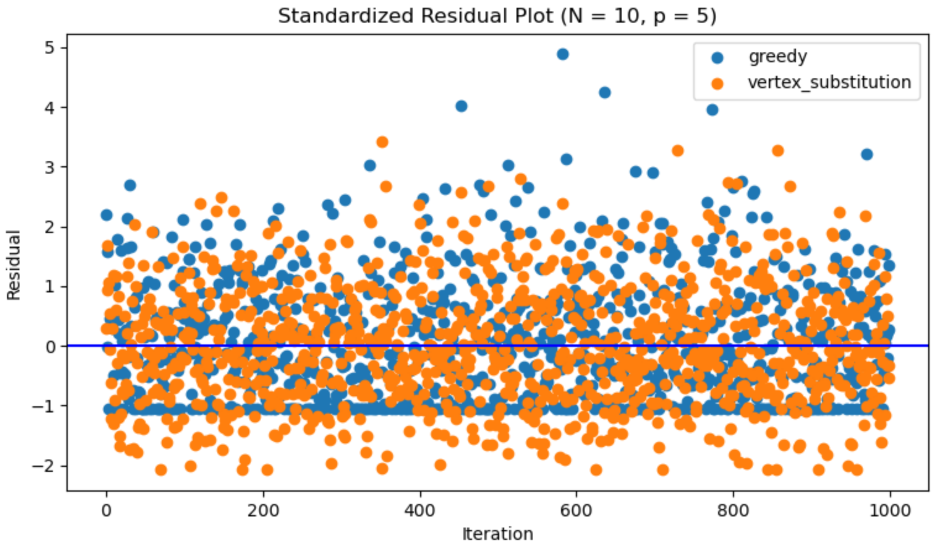


Figure 5: Standardized residuals for the heuristics when $N = 10$ and $p = 5$

of accuracy mathematically. "Confidence Level" estimation is a technique that can be employed to check which heuristic is mathematically closer. This will be explored in the future (See section 6.1).

5 Real World Application

5.1 Problem Definition

We have identified 10 Nandini Milk Parlors across the Bangalore Sales Depot Area. All of these Parlors must be serviced daily with one shipment. The objective of the problem is to find the ideal number of factories(medians) and their location to set up to minimise the registry cost and delivery cost of milk products. We assume all nodes have equal demand and receive the same quantity and type of shipment, provided by the factories. The registry cost is borne every year, and hence the total cost for one entire year must be taken into consideration. This is an application of The Uncapacitated Problem.

$$
D = \begin{bmatrix}
0 & 4.76 & 5.48 & 8.4 & 7.64 & 10.2 & 4.11 & 3.71 & 5.12 & 4.21 \\
4.76 & 0 & 1.23 & 5.08 & 5.50 & 11.3 & 4.89 & 2.45 & 3.99 & 4.27 \\
5.48 & 1.23 & 0 & 3.83 & 4.36 & 10.5 & 4.51 & 2.26 & 3.22 & 3.76 \\
8.4 & 5.08 & 3.83 & 0 & 1.82 & 8.98 & 5.51 & 4.68 & 3.60 & 4.76 \\
7.64 & 5.50 & 4.36 & 1.82 & 0 & 7.27 & 4.32 & 4.19 & 2.61 & 3.53 \\
10.2 & 11.3 & 10.5 & 9.98 & 7.27 & 0 & 6.77 & 9.13 & 7.44 & 7.21 \\
4.11 & 4.89 & 4.51 & 5.51 & 4.32 & 6.77 & 0 & 2.61 & 1.96 & 0.88 \\
3.71 & 2.45 & 2.26 & 4.68 & 4.19 & 9.13 & 2.61 & 0 & 1.87 & 1.77 \\
5.12 & 3.99 & 3.22 & 3.60 & 2.61 & 7.44 & 1.96 & 1.87 & 0 & 1.13 \\
4.21 & 4.27 & 3.76 & 4.76 & 3.53 & 7.21 & 0.88 & 1.77 & 1.13 & 0
\end{bmatrix}
$$

$$
C = \begin{pmatrix} 4400 & 10100 & 10100 & 4400 & 4400 & 6700 & 4710 & 4710 & 4710 & 4710 \end{pmatrix}
$$

The distances here are in *kilometers* and are calculated using the **Google My Maps** (https://www.google.com/maps/d/u/0/) platform. The costs above are the *average* registry cost per square ft. at each parlor in Indian Ruppees.

Figure 6 visually shows the parlors selected in the Bangalore Sales Depot Area. The index number in the figure corresponds to the row/ column number in D and C.

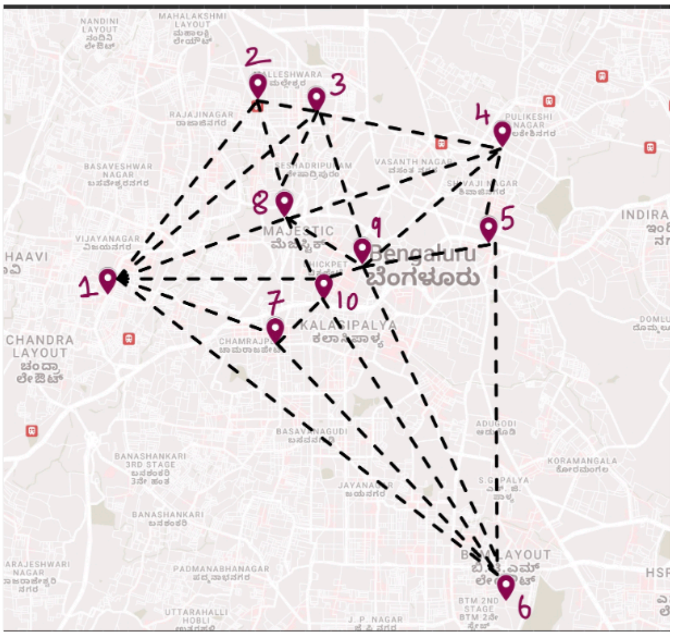


Figure 6: Real World Application Definition

5.2 Code Output

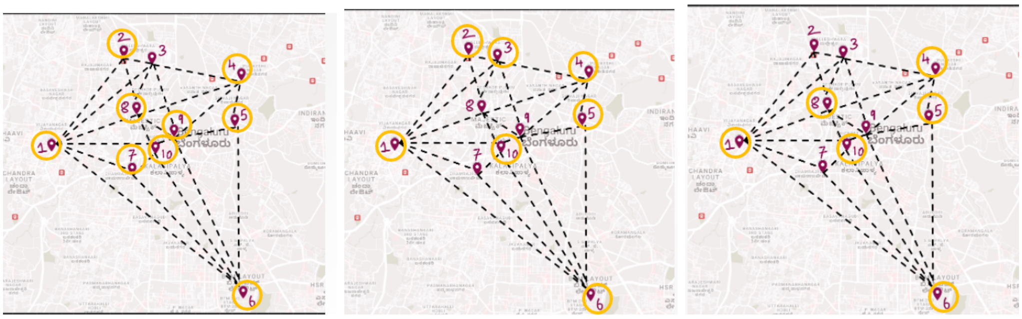
Solving the real-world application using the three algorithms above gives us the following results:

Greedy Selected Medians: [3, 0, 5, 1, 4, 6, 7, 8, 9] (9 Medians)
Cost: 50840.0

Enumerate Selected Medians: [0, 3, 4, 5, 7, 9] (6 Medians)
Cost: 47716.0

Vertex Substitution Selected Medians: [0, 1, 2, 3, 4, 5, 9] (7 Medians)
Cost: 47890.0

The selected medians can be visualized in the graphs present in Figure 7.



(a) Greedy Heuristic (b) Vertex Substitution (c) Enumeration Solver

Figure 7: Solutions to the Real World Problem (Selected Heuristics circled in Yellow)

5.3 Inference of Output and Conclusions

The Greedy Approach selected 9 medians (factory locations), resulting in a total cost of 50840.0. Vertex Substitution selected fewer number of nodes, and had a lower total cost as compared to the Greedy approach. The Enumerate Approach chose 6 medians with a total cost of 47716.0. The greedy approach might not guarantee the optimal solution but provides a quick solution. The enumeration approach is the most exhaustive, evaluating different combinations to minimize the cost. The Vertex Substitution heuristic generally provides an in-between ground not sacrificing the accuracy and also having a quick runtime as compared to enumeration. This result is consistent with the statistical comparisions conducted in Section 4.

To decide between the the different approaches, it's crucial to balance computational complexity with the need for an *optimal* or *near-optimal* solution. The enumeration method might be more accurate but can quickly become impractical for larger datasets due to its exhaustive nature.

6 Conclusion

This Project has been a cursory dive into the world of heuristic methods to solve the p-median facility allocation problem. We can conlude that choosing a heuristic depends on several factors, most importantly the context of where the solution needs to be used.

If the use-case is critical, then perhaps a heuristic cannot be used at all. In the mission to reduce computational complexity, it is important to note that accuracy always reduces.

Overall, the project has showed us how to use different methods to solve the same problem, and the brute-force approach is almost never the answer. The following sub-section outlines the future scope of this project, which we plan to continue and convert into a research paper if the circumstances let us do so. There are several avenues left to explore.

6.1 Future Scope

- Understand the Double Vertex Substitution Method. While researching, we came across the "Double Vertex Substitution Heuristic" which is an extension of the Vertex Substitution Heuristic. We would like to exploer this in further detail to find out it's importance [4].
- Try to establish **confidence level** for heuristic algorithms, understanding when they work and when they don't mathematically rather than only experimentally.
- Explore integrating machine learning for adaptive decision-making with dynamic datasets. Can Machine Learning provide a method to choose the right heuristic?
- Investigating hybrid solutions combining the strengths of Greedy, Vertex Substitution and Enumerate methods for flexibility - Creating a new heuristic, particularly for the Capacitated Problem.
- Implement frameworks for continuous adaptation and improvement based on performance feedback and changing requirements and hopefully write a research paper in the future!

References

[1] Ramzi Al-Aruri. *ralaruri/p_median_python*. GitHub. May 2023. URL: https://github.com/ralaruri/p%5C_median%5C_python.

[2] A. Aswani. *IEOR 151 lecture 13 P-median problem - University of California, Berkeley*. URL: https://aswani.ieor.berkeley.edu/teaching/FA16/151/lecture_notes/ieor151%5C_lec13.pdf.

[3] Design and Analysis of Supply Chains. *Greedy Heuristic for Solving the Set Covering Problem*. Youtube. Sept. 15, 2020. URL: https://www.youtube.com/watch?v=Hw_IAYqtrLI.

[4] Samuel Eilon and Roberto D. Galvão. "Single and Double Vertex Substitution in Heuristic Procedures for the p-Median Problem". In: *Management Science* 24.16 (1978), pp. 1763–1766. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2630474 (visited on 12/17/2023).

[5] Alfred A. Kuehn and Michael J. Hamburger. "A Heuristic Program for Locating Warehouses". In: *Management Science* 9.4 (1963), pp. 643–666. ISSN: 00251909, 15265501.

[6] nptelhrd. *Mod-08 Lec-32 Location problems – p median problem, Fixed charge problem*. Youtube. June 3, 2022. URL: https://www.youtube.com/watch?v=cHV4oZEODyo.

[7] IITMadras - B.S. Degree Programme. *Facility Location Problem Tutorial*. Youtube. Sept. 15, 2022. URL: https://www.youtube.com/watch?v=bdy2qg6dwac.

[8] Zach Siegel. *P-Median Problems and Solution Strategies*. Dec. 2021. URL: https://zsiegel92.github.io/writing%5C_repo/UCLA/mgmt242/pmedian.pdf.

[9] Michael B. Teitz and Polly Bart. "Heuristic Methods for Estimating the Generalized Vertex Median of a Weighted Graph". In: *Operations Research* 16.5 (1968), pp. 955–961. DOI: 10.1287/opre.16.5.955. eprint: https://doi.org/10.1287/opre.16.5.955. URL: https://doi.org/10.1287/opre.16.5.955.

[10] Ningchuan Xiao. *GIS Algorithms: Theory and Applications for Geographic Information Science & Technology*. 55 City Road, London, Dec. 2023. DOI: 10.4135/9781473921498. URL: https://sk.sagepub.com/books/gis-algorithms.

Appendix A: Supplemental Materials

Code

The code associated with the project is written in `Python 3.11` and requires the following requirements to run:

```
matplotlib
numpy
```

It is arranged across 4 files:

`main.py` Contains the problem definitions and code to run them.
`solvers.py` Contains the bulk of the code for implementing the heuristic algorithms
`visualization.py` Contains code for comparing heuristics
`utils.py` Contains some helpful functions for other files.

Updated code with up-to-date in line documentation is available on GitHub at `https://github.com/PjrCodes/pmedian_opt`.

File main.py

```python
from solvers import *
import copy

def uncapacitated_problem(
    distance_matrix, cost_array, node_count, solver="greedy"
) -> tuple[list[int], float]:
    best_cost = np.inf
    best_medians = []
    best_p = 0
    opcount = 0

    for p in range(2, node_count + 1):
        if solver == "greedy":
            selected_medians, no_ops = greedy_solver(
                distance_matrix, cost_array, node_count, p
            )
        elif solver == "enumeration":
            selected_medians, no_ops = enumeration_solver(
```

```
                distance_matrix, cost_array, node_count, p
            )
        elif solver == "vertex_substitution":
            # print(f"For {p=}")
            selected_medians, no_ops = vertex_substitution_solver(
                distance_matrix, cost_array, node_count, p
            )
            # print("\n")
        else:
            raise ValueError(f"Unknown solver: {solver}")

        if selected_medians is not None:
            # we were able to find a solution
            total_cost = cost_of_configuration(
                distance_matrix, cost_array, node_count, selected_medians
            )
            if total_cost < best_cost:
                best_cost = total_cost
                best_medians = selected_medians
                best_p = p
                opcount += no_ops
        else:
            print(f"\tUnable to find a solution for {p=}")

    return best_cost, best_medians, best_p, opcount


def capacitated_problem(
    distance_matrix,
    cost_array,
    node_count,
    production_array,
    demand_array,
    solver="greedy",
):
    best_cost = np.inf
    best_medians = []
    best_p = 0
    opcount = 0

    # Iterating for p from 1 to node_count, and finding medians with least
    ↪  cost #
    for p in range(1, node_count + 1):
        if solver == "greedy":
            selected_medians, no_ops = greedy_like(
                distance_matrix,
                cost_array,
                node_count,
```

```
                production_array,
                demand_array,
                p,
            )
        else:
            raise ValueError(f"Unknown solver: {solver}")

        if selected_medians is not None:
            total_cost = cost_of_configuration_for_variantB(
                distance_matrix,
                selected_medians,
                node_count,
                cost_array,
                production_array,
                demand_array,
            )
            if total_cost < best_cost:
                best_cost = total_cost
                best_medians = selected_medians
                best_p = p
                opcount += no_ops
        else:
            print(f"\tUnable to find a solution for {p=}")

    return best_cost, best_medians, best_p, opcount


def real_world_input():
    Dij = np.array(
        [
            [0.00, 4.76, 5.48, 8.4, 7.64, 10.20, 4.11, 3.71, 5.12, 4.21],
            [4.76, 0.00, 1.23, 5.08, 5.50, 11.3, 4.89, 2.45, 3.99, 4.27],
            [5.48, 1.23, 0.00, 3.83, 4.36, 10.5, 4.51, 2.26, 3.22, 3.76],
            [8.40, 5.08, 3.83, 0.00, 1.82, 8.98, 5.51, 4.68, 3.69, 4.76],
            [7.64, 5.50, 4.36, 1.82, 0.00, 7.27, 4.32, 4.19, 2.61, 3.53],
            [10.2, 11.3, 10.5, 9.98, 7.27, 0.00, 6.77, 9.13, 7.44, 7.21],
            [4.11, 4.89, 4.51, 5.51, 4.32, 6.77, 0.00, 2.61, 1.96, 0.88],
            [3.71, 2.45, 2.26, 4.68, 4.19, 9.13, 2.61, 0.00, 1.87, 1.77],
            [5.12, 3.99, 3.22, 3.60, 2.61, 7.44, 1.96, 1.87, 0.00, 1.13],
            [4.21, 4.27, 3.76, 4.76, 3.53, 7.21, 0.88, 1.77, 1.13, 0.00],
        ]
    )
    D_cost = Dij * 7.5 * 365
    C = np.array([4400, 10100, 10100, 4400, 4400, 6700, 4710, 4710, 4710, 4710])
    node_count = 10

    return D_cost, C, node_count
```

```


if __name__ == "__main__":
    # distance_matrix = np.array(
    #     [
    #         [0, 5, 100, 4, 5],
    #         [5, 0, 5, 5, 100],
    #         [100, 5, 0, 5, 5],
    #         [4, 5, 5, 0, 5],
    #         [5, 100, 5, 5, 0],
    #     ]
    # )

    # cost_array = np.array([1, 1, 1, 1, 1])
    # node_count = 5
    # demand_array = np.array([10, 15, 25, 15])
    # production_array = np.array([5, 7, 6, 5])
    # print(vertex_substitution_solver(distance_matrix, cost_array,
    ↪  node_count, 2))

    # median, _ = vertex_substitution_solver(distance_matrix, cost_array,
    ↪  node_count, 2)
    # print(median)
    # print(cost_of_configuration(distance_matrix, cost_array, node_count,
    ↪  median))

    # median, _ = greedy_solver(distance_matrix, cost_array, node_count, 2)
    # print(median)
    # print(cost_of_configuration(distance_matrix, cost_array, node_count,
    ↪  median))
    # D, C, n = real_world_input()
    # Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(n)
    # print(
    #     "Vertex Substitution:",
    #     uncapacitated_problem(
    #         Di, Ci, ni, solver="vertex_substitution"
    #     ),
    # )
    # Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(n)
    # print(
    #     "Greedy:",
    #     uncapacitated_problem(Di, Ci, ni, solver="greedy"),
    # )
    # Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(n)
    # print(
    #     "Enumeration:",
    #     uncapacitated_problem(
    #         Di, Ci, ni, solver="enumeration"
```

```
    #     ),
    # )

    # D = np.array([[0, 5, 10], [5, 0, 7], [10, 7, 0]])
    # C = np.array([1, 2, 1])
    # W = np.array([4, 6, 5])
    # P = np.array([1.5, 2, 1.7])
    # NC = 3

    # cost, selected_medians, p, no_ops = capacitated_problem(
    #     D, C, NC, P, W, solver="greedy"
    # )
    # print(f"{cost=} {selected_medians=} {p=} {no_ops=}")

    D = np.array([[0, 5, 10], [5, 0, 7], [10, 7, 0]])
    C = np.array([5, 7, 5])
    NC = 3
    # print(f"{D=}\n{C=}\n{NC=}\n")
    # best_cost, best_medians, best_p, opcount
    cost, selected_medians, p, no_ops = uncapacitated_problem(
        D, C, NC, solver="vertex_substitution"
    )
    # print(f"{cost=} {selected_medians=} {p=} {no_ops=}")

    # cost, selected_medians, p, no_ops = uncapacitated_problem(
    #     D, C, NC, solver="enumeration"
    # )
    # print(f"{cost=} {selected_medians=} {p=} {no_ops=}")
```

File solvers.py

```
import itertools
import numpy as np


def cost_of_configuration(
    distance_matrix: np.ndarray,
    cost_array: np.ndarray,
    node_count: int,
    median_list: np.ndarray,
) -> float:
    """Generic cost calculation function used for comparisions."""

    cost = 0
```

```python
    for point in range(node_count):
        mindist = float("inf")
        for median in median_list:
            dist = distance_matrix[point, median]
            if dist < mindist:
                mindist = dist
        cost += mindist

    cost += sum([cost_array[median] for median in median_list])

    return cost


def calculate_cost_for_greedy(
    distance_matrix: np.ndarray, cost_array, median_idxs: list[int]
) -> float:
    """Cost calculation algorithm used for greedy."""

    visited_median_idxs = []
    cost = 0
    for median_idx in median_idxs:
        dist_arr = distance_matrix[:, median_idx]
        # print(f"{median_idx=} {dist_arr=}")
        # distances of MEDIAN to all points as below
        facility_cost_of_median = cost_array[median_idx]

        # Distance to Visited Medians Must not be considered#
        for i in visited_median_idxs:
            dist_arr[i] = 0

        # print(f"{dist_arr=}")
        # print(f'{median_idx=},{visited_median_idxs=},{dist_arr=}')
        cost += np.sum(dist_arr) + facility_cost_of_median
        visited_median_idxs.append(median_idx)

    return cost


def greedy_solver(
    distance_matrix: np.ndarray, cost_array: np.ndarray, node_count: int, p:
    ↪ int = 1
) -> tuple[list[int], int]:
    no_ops = 0

    if p >= node_count:
        return list(range(node_count)), no_ops + 1  # select all and return

    best_median_idxs: list[int] = []
```

```python
    remaining_point_idxs = np.arange(node_count)

    for _ in range(p):
        best_cost = np.inf
        best_median_idx = None

        for node_idx in remaining_point_idxs:
            no_ops += 1
            # node_idx is the index that I am currently considering as a median(s)
            candidate_positions_for_medians = best_median_idxs.copy() + [node_idx]

            cost = calculate_cost_for_greedy(
                distance_matrix, cost_array, candidate_positions_for_medians
            )

            if cost < best_cost:
                best_cost = cost
                best_median_idx = node_idx

        best_median_idxs.append(best_median_idx)

        # delete the selected median from the candidate list
        remaining_point_idxs = np.delete(
            remaining_point_idxs, np.where(remaining_point_idxs == best_median_idx)
        )

    return best_median_idxs, no_ops


def vertex_substitution_solver(D, cost_array, node_count: int, p: int = 2):
    V = np.arange(node_count)
    R = np.zeros((node_count, node_count))
    i = 0
    # for cost in cost_array:
    for j in V:
        R[:, j] = cost_array[j] * D[:, j]
        # i += 1

    # print(R)

    V_cand = np.copy(V[:p])

    no_ops = 1

    previous = -1000
```

```
    cost = 0
    while True:
        # print(f"{'='*20} Cycle {no_ops} {'='*20}")

        # print(f"V1 = {V_cand}")
        # Inside a cycle
        partitions = []

        for jvertex in V_cand:
            part = []
            for ivertex in V:
                r_ij = R[ivertex, jvertex]
                for kvertex in V_cand:
                    r_ik = R[ivertex, kvertex]
                    if r_ij > r_ik:
                        break
                else:
                    part.append(ivertex)
            partitions.append(part)

        # print(f"{partitions=}")

        # ---- Vb repeat ---- #
        seen = set(V_cand)
        rem = set(V) - seen
        vbidx = 0
        while vbidx < len(rem):
            SubR = R[:, V_cand]
            # print(f"Submatrix Rp = {SubR}")
            Vb = rem.pop()
            rem.add(Vb)  # temp, later removed!
            # print(f"Trying to substitute {Vb} inside current set {V_cand}.")
            deltabjs = []
            for jvertexidx in np.arange(len(V_cand)):
                # print(f"Trying to substitute {V_cand[jvertexidx]} with
                ↪  {Vb}.")

                # replace jvertex with Vb
                V_cand_copy = V_cand.copy()
                V_cand_copy[jvertexidx] = Vb
                # print(f"Trying {V_cand_copy}")
                i = 0
                delta = 0
                for ithrow in SubR:
                    minimum = np.min(ithrow)
                    second_minimum = np.sort(ithrow)[1]
                    rij = SubR[i, jvertexidx]
                    delt = 0
```

```
                    if minimum == rij:
                        if R[i, Vb] <= rij:
                            delt = R[i, Vb] - rij  # shold be negative
                            # print(f"delt: {delt}")
                        elif rij <= R[i, Vb] and R[i, Vb] <= second_minimum:
                            delt = R[i, Vb] - rij  # should be positive
                        elif rij <= R[i, Vb] and second_minimum <= R[i, Vb]:
                            delt = second_minimum - rij  # should be positive
                    # print(f"For row {i=}, ith delta = {delt}")
                    delta += delt
                    i += 1
                # print(f"Delta for configuration = {delta}")
                deltabjs.append(delta)

            bjmin = np.min(deltabjs)
            if bjmin < 0:
                # print(f"Found a good replacement, {V_cand[np.argmin(deltabjs)]} -> {Vb}")
                V_cand[np.argmin(deltabjs)] = Vb
            else:
                pass
                # print(f"Did not find any good replacements.")

            seen.add(Vb)
            rem = set(V) - seen
            vbidx += 1
        # ---- Vb repeat ---- #

        # cost #
        cost = 0
        for j in np.arange(len(V_cand)):
            for i in partitions[j]:
                cost += R[i, V_cand[j]]

        # print(f"VS Cost = {cost}")
        if cost == previous:
            # print(f"Cost is same as last cycle, ending computation.")
            break
        else:
            previous = cost
        if no_ops > 10000:
            break

        no_ops += 1

    return V_cand, no_ops
```

```python
def enumeration_solver(
    distance_matrix, cost_array, node_count: int, p: int = 1
) -> list[int]:
    """Enumeration method for the p-median problem."""

    best_cost = float("inf")
    best_median_idxs = []
    no_ops = 0

    # Enumerate all possible combinations of p medians
    for candidate_median_idxs in itertools.combinations(np.arange(node_count), p):
        no_ops += 1

        cost = cost_of_configuration(
            distance_matrix, cost_array, node_count, candidate_median_idxs
        )

        if cost < best_cost:
            best_cost = cost
            best_median_idxs = candidate_median_idxs

    return best_median_idxs, no_ops


def cost_of_configuration_for_variantB(
    distance_matrix, median_idxs, node_count, cost_array, production_array, demand_array
) -> float:
    # Generating Delivery Weight Matrix #
    delivery_weight = np.zeros((node_count, node_count))
    for node in range(node_count):
        for median in range(node_count):
            delivery_weight[node, median] = (
                demand_array[node] / production_array[median]
            ) * cost_array[median]

    weighted_distance_matrix = delivery_weight + distance_matrix

    # Generating X Matrix based on minimum distance
    X = np.zeros((node_count, node_count))
    for i in range(node_count):
        # If i is already a median, skip #
        if i in median_idxs:
            continue

        # Initializing Minimum Distance and Best Median
        mindist = float("inf")
```

```python
        best_median = None

        # Iterating though Medians to find Closest Median
        for median in median_idxs:
            dist = weighted_distance_matrix[median][i]
            if dist < mindist and median != i:
                mindist = dist
                best_median = median

        # Assigning Node to Closest Median
        X[best_median, i] = 1

    # Finding Facility and Delivery Cost
    facility_cost = 0
    delivery_cost = 0
    for median in median_idxs:
        median_demand = np.dot(demand_array, X[median]) + demand_array[median]
        facility_area = median_demand / production_array[median]
        facility_cost += facility_area * cost_array[median]

        delivery_cost += np.dot(X[median], distance_matrix[median])

    return delivery_cost + facility_cost


def calculate_cost_for_greedy_for_variantB(
    median_idxs,
    distance_matrix,
    node_count: int,
    cost_array,
    production_array,
    demand_array,
) -> float:
    # Defining Visted Medians, Cost, and Total Demand #
    visited_median_idxs = []
    cost = 0
    total_demand = np.sum(demand_array)

    for median_idx in median_idxs:
        # Calculating Distance of Median to Remaining Nodes #
        total_distance = 0
        median_cost = 1
        for i in range(node_count):
            if i not in visited_median_idxs:
                total_distance += distance_matrix[median_idx, i]

        # Dividing Demand to be Fulfilled by Production per unit area #
        area = total_demand / production_array[median_idx]
```

```python
        median_cost = area * cost_array[median_idx]

        # Adding Cost of median to Total Cost #
        cost += median_cost + total_distance

        # Removing Demand of Median from Total_Demand #
        total_demand -= demand_array[median_idx]

        # Appending Current Median to Visited Medians #
        visited_median_idxs.append(median_idx)

    return cost


def greedy_like(
    distance_matrix,
    cost_array,
    node_count: int,
    production_array,
    demand_array,
    p: int = 1,
) -> list[int]:
    no_ops = 0

    # If p >= nodes, select all nodes
    if p >= node_count:
        return list(range(node_count)), no_ops + 1

    # Initializing Best Medians and Remaining Points #
    best_median_idxs: list[int] = []
    remaining_point_idxs = np.arange(node_count)

    # For each p, finding lowest cost median, appending to best_median_idxs
    ↪  and removing median from remaining_point_idxs
    for _ in range(p):
        best_cost = np.inf

        for node_idx in remaining_point_idxs:
            no_ops += 1
            # node_idx is the index that I am currently considering as a
            ↪  median(s)
            candidate_positions_for_medians = best_median_idxs.copy() +
            ↪  [node_idx]

            cost = calculate_cost_for_greedy_for_variantB(
                candidate_positions_for_medians,
                distance_matrix,
                node_count,
```

```
                cost_array,
                production_array,
                demand_array,
            )

            if cost < best_cost:
                best_cost = cost
                best_median_idx = node_idx

        best_median_idxs.append(best_median_idx)

        # delete the selected median from the candidate list
        remaining_point_idxs = np.delete(
            remaining_point_idxs, np.where(remaining_point_idxs ==
            ↪  best_median_idx)
        )

    return best_median_idxs, no_ops
```

File visualization.py

```
import copy
from datetime import datetime
import matplotlib.pyplot as plt
import numpy as np
from main import uncapacitated_problem, real_world_input

from solvers import cost_of_configuration, enumeration_solver, greedy_solver,
↪  vertex_substitution_solver

def input_creator(N):
    # D is a randomly generated NxN symmetric matrix
    D = np.random.randint(0, 500, size=(N, N))
    D = (D + D.T) / 2

    # C is a randomly generated N array
    C = np.random.randint(1, 1000, size=N)
    # rand = np.random.randint(1, 1000)
    # C = rand * np.ones(N)

    return D, C


def plotter(N, TIME_greedy, TIME_tb, TIME_es):
```

```
    TIME_greedy_avgs = []
    TIME_tb_avgs = []
    TIME_es_avgs = []
    for p in range(2, N):
        TIME_greedy_avgs.append(np.mean(TIME_greedy[p]))
        TIME_tb_avgs.append(np.mean(TIME_tb[p]))
        TIME_es_avgs.append(np.mean(TIME_es[p]))

    plt.plot(list(range(2, N)), TIME_greedy_avgs, label="greedy")
    plt.plot(list(range(2, N)), TIME_tb_avgs, label="vertex_substitution")
    plt.plot(list(range(2, N)), TIME_es_avgs, label="enumeration")

    plt.legend()
    plt.title(f"Time Comparision (N = {N})")
    plt.xticks(list(range(2, N)))
    plt.xlabel("p")
    plt.ylabel("Time (microseconds)")
    plt.show()

def comparator(N):
    TIME_greedy = [[] for _ in range(N)]
    TIME_tb = [[] for _ in range(N)]
    TIME_es = [[] for _ in range(N)]

    for p in range(2, N):
        for i in range(50):
            print(f"{N=}, {p=}, {i=}")

            D, C = input_creator(N)
            actp = p + 1
            Di, Ci = copy.deepcopy(D), copy.deepcopy(C)

            st = datetime.now()
            greedy_solver(Di, Ci, N, actp)
            ttg = datetime.now() - st
            Di, Ci = copy.deepcopy(D), copy.deepcopy(C)

            st = datetime.now()
            vertex_substitution_solver(Di, Ci, N, actp)
            tttb = datetime.now() - st
            Di, Ci = copy.deepcopy(D), copy.deepcopy(C)

            st = datetime.now()
            enumeration_solver(Di, Ci, N, actp)
            ttes = datetime.now() - st

            TIME_greedy[p].append(ttg.microseconds)
            TIME_tb[p].append(tttb.microseconds)
```

```
            TIME_es[p].append(ttes.microseconds)

    return TIME_greedy, TIME_tb, TIME_es


def visualize_each_p():
    # from utils import cost_of_configuration
    D, C, node_count = real_world_input()

    greedy_times = []
    tb_times = []
    es_times = []


    for i in range(1000):
        Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C),
        ↪  copy.deepcopy(node_count)
        st = datetime.now()
        uncapacitated_problem(Di, Ci, ni, solver="greedy")
        greedy_times.append(datetime.now() - st)
        Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C),
        ↪  copy.deepcopy(node_count)
        st = datetime.now()
        uncapacitated_problem(Di, Ci, ni, solver="vertex_substitution")
        tb_times.append(datetime.now() - st)
        Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C),
        ↪  copy.deepcopy(node_count)
        st = datetime.now()
        uncapacitated_problem(Di, Ci, ni, solver="enumeration")
        es_times.append(datetime.now() - st)

    return np.mean(greedy_times), np.mean(tb_times), np.mean(es_times)
    # print(f"{greedy_time=}\n{tb_time=}\n{es_time=}")

def visualize_each_n(p = 5, Nmax = 21):
    # from utils import cost_of_configuration
    TIME_greedy = [[] for _ in range(Nmax)]
    TIME_tb = [[] for _ in range(Nmax)]
    # TIME_es = [[] for _ in range(Nmax)]

    avgs_g = []
    avgs_tb = []
    # avgs_es = []

    for N in range(3, Nmax):
        D, C = input_creator(N)

        for i in range(100):
```

```
            Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(N)
            st = datetime.now()
            greedy_solver(D, C, N, p)
            TIME_greedy[N].append((datetime.now() - st).microseconds)
            Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(N)
            st = datetime.now()
            vertex_substitution_solver(D, C, N, p)
            TIME_tb[N].append((datetime.now() - st).microseconds)
            Di, Ci, ni = copy.deepcopy(D), copy.deepcopy(C), copy.deepcopy(N)
            # st = datetime.now()
            # enumeration_solver(D, C, N, p)
            # TIME_es[N].append((datetime.now() - st).microseconds)
            print(f"{N=}, {i=}")

        avgs_g.append(np.mean(TIME_greedy[N]))
        avgs_tb.append(np.mean(TIME_tb[N]))
        # avgs_es.append(np.mean(TIME_es[N]))

    print(avgs_g, end="\n\n")
    print(avgs_tb, end="\n\n")
    # print(avgs_es, end="\n\n")

    # p = 5
    # avgs_g = [0.83, 0.65, 0.73, 140.91, 167.68, 198.78, 232.1, 275.05,
    ↪  314.36, 346.54, 410.7, 434.5, 481.31, 514.83, 549.0, 582.02, 614.77,
    ↪  652.85]
    # avgs_tb = [26.37, 36.64, 52.16, 262.03, 284.75, 561.78, 621.15, 975.14,
    ↪  1567.64, 1505.18, 2394.58, 3051.89, 3374.13, 2826.44, 4351.62,
    ↪  5327.13, 5618.67, 6718.34]
    # avgs_es = [1.48, 1.46, 6.54, 33.45, 120.55, 353.52, 866.24, 1919.46,
    ↪  3796.39, 6915.65, 12252.08, 20420.13, 32480.14, 49786.66, 75983.48,
    ↪  110364.27, 158368.82, 221628.27]

    plt.plot(list(range(3, Nmax)), avgs_g, label="greedy")
    plt.plot(list(range(3, Nmax)), avgs_tb, label="vertex_substitution")
    # plt.plot(list(range(3, 21)), avgs_es, label="enumeration")
    plt.legend()
    plt.title(f"Time Comparision (p = {p})")
    plt.xticks(list(range(3, Nmax)))
    plt.xlabel("N")
    plt.ylabel("Time (microseconds)")
    plt.savefig("time_comparison_nchange.png")
    plt.show()



def plot_bar(time_greedy, time_tb, time_es):

```

```
    plt.bar(0, time_greedy.microseconds, label="greedy")
    plt.bar(1, time_tb.microseconds, label="greedy")
    plt.bar(2, time_es.microseconds, label="greedy")
    plt.title("Time Comparision (for Real World Problem)")
    plt.xlabel("Solver")
    plt.xticks([0, 1, 2], ["Greedy", "Vertex Substitution", "Enumeration"])
    plt.ylabel("Time (microseconds)")
    # plt.legend()
    plt.show()

def residual_plot(N=10, p=5, imax=100):

    diff_greedy = []
    diff_tb = []

    for i in range(imax):
        print(f"{i=}")
        D, C = input_creator(N)
        Di, Ci = copy.deepcopy(D), copy.deepcopy(C)
        best_median_idxs, opc = greedy_solver(Di, Ci, N, p)
        costG = cost_of_configuration(D, C, N, best_median_idxs)

        Di, Ci = copy.deepcopy(D), copy.deepcopy(C)
        best_median_idxs, opc = vertex_substitution_solver(Di, Ci, N, p)
        costTB = cost_of_configuration(D, C, N, best_median_idxs)

        Di, Ci = copy.deepcopy(D), copy.deepcopy(C)
        best_median_idxs, opc = enumeration_solver(Di, Ci, N, p)
        costENUM = cost_of_configuration(D, C, N, best_median_idxs)

        diff_greedy.append(costG - costENUM)
        diff_tb.append(costTB - costENUM)

    # standardize the data
    diff_greedy = np.array(diff_greedy)
    diff_tb = np.array(diff_tb)
    diff_greedy = (diff_greedy - np.mean(diff_greedy)) / np.std(diff_greedy)
    diff_tb = (diff_tb - np.mean(diff_tb)) / np.std(diff_tb)


    plt.scatter(list(range(imax)), diff_greedy, label="greedy")
    plt.scatter(list(range(imax)), diff_tb, label="vertex_substitution")
    plt.legend()
    plt.title(f"Standardized Residual Plot (N = {N}, p = {p})")
    plt.xlabel("Iteration")
    plt.ylabel("Residual")
    plt.axhline(0, color="blue")
    plt.show()
```

```python

if __name__ == "__main__":
    # N = 15
    # TIME_greedy, TIME_tb, TIME_es = comparator(N)
    # print("\n\n\n")
    # print(f"{TIME_tb=}\n\n{TIME_es=}\n\n{TIME_greedy=}\n")
    # print("\n\n\n")
    # plotter(N, TIME_greedy, TIME_tb, TIME_es)
    # plot_bar(*visualize_each_p())
    # visualize_each_n(p=3, Nmax=50)
    residual_plot(p=5, imax=1000)
```

File utils.py

```python
import numpy as np

def generate_random_inputs(n):
    """
    A function that generates the random attributes for a Variant A problem.

    @arguments
    node_count: int - the number of nodes.

    @returns
    distance_matrix - distances from node i to node j
    cost_array - cost per unit area for each median
    """

    # Generating a random symmetric matrix
    matrix = np.random.rand(n, n)
    distance_matrix = (matrix + matrix.T) / 2  # Ensuring symmetry

    # Fill diagonal entries with zeros
    np.fill_diagonal(distance_matrix, 0)

    cost_matrix = np.random.rand(n)

    return distance_matrix, cost_matrix, n
```