LS_Linear_Optimzation

April 30, 2024

1 LINEAR OPTIMIZATION FINAL PROJECT

2 ROBUST SHORTEST TIME DELIVERY PATHS

The Amazon company has a startup project specializing in package delivery within the western areas of the United States. They advertise guaranteed delivery within a specified time. Their quoted times are not always shorter than other delivery companies, but delivery comes with a money-back guarantee if, for any reason, the delivery is not met on time. The company necessarily has a vested interest in finding delivery routes that are not only efficient (short) but robust (not likely to incur delays). In order to set their rates, the company is seeking a method for finding an optimal route between two given locations.

The problem of finding a shortest path between two locations can be easily solved as an integer program, provided that the number of possible routes is not too large. Also,the problem of finding the route least likely to incur a delay is similarly solved. Amazon would like to solve the problem of a compromise route that helps then maintain competitiveness without significant reduction in promised delivery time.

You are given the following tasks:

1. Use the provided network data to solve the shortest path problem for user-supplied starting and ending locations. Ignore any possible delays.
2. Use the provided network data to solve the minimum delay likelihood problem for user-supplied starting and ending locations. Ignore travel times.
3. Combine the two previous solution methods where the objective is to minimize the weighted sum of travel time and delay likelihood. The relative weight should be an adjustable parameter. Determine a reasonable relative weighting based on experiments you perform.
4. Solve the combined problem using your weight selection with starting city A and ending locations at every other city. Report your findings and make recommendations to the company.

The data set is provided as two files distance.csv and delay.csv which are commadelimited 15 x 15 arrays. The entry in the ith row and jth column of the respective arrays gives the distance (or delay probability) between city i and city j. If an array has entry zero, this indicates that the cities are not connected by a path.

The networks we will consider are symmetric - distances and delays do not depend on the direction of travel.

3 MODELING THE PROBLEM

Let x_{ij} be a binary decision variable where:

$x_{ij} = 1$ if the path from city i to city j is included in the travel from city A to city O , and $x_{ij} = 0$ otherwise.

The objective function to minimize is the total distance of the path:

$$\text{Minimize} \sum_{i,j} d_{ij} \cdot x_{ij} \tag{1}$$

Subject to the following constraints:

Ensure that there is zero or one outgoing edge from each inner city(except for A, O):

$$\sum_{j} x_{ij} = y_j \quad \text{for} \quad i \in \{B, C, D, E, F, G, H, I, J, K, L, M, N\} \tag{2}$$

Ensure that there is zero or one incoming edge to each inner city (except for A, O):

$$\sum_{i} x_{ij} = y_i \quad \text{for} \quad j \in \{B, C, D, E, F, G, H, I, J, K, L, M, N\} \tag{3}$$

where $y_j \in \{0, 1\}$, and $y_i \in \{0, 1\}$. Note that although it seems we used different notations for the number of outgoing and incoming edges from a city, they (y_j and y_i) are the same numbers for the same city. For example, if the number of outgoing edges from city B ($j = B$) is y_B, so is that of incoming edges to city B ($i = B$), which means y_j and y_i are the same numbers for the same city.

Ensure that the destination city has exactly zero outgoing edges:

$$\sum_{j} x_{Oj} = 1 \quad \text{for} \quad j \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\} \tag{4}$$

Ensure that the starting city has exactly zero incoming edges:

$$\sum_{i} x_{iA} = 0 \quad \text{for} \quad i \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\} \tag{5}$$

Ensure that the destination city has exactly one incoming edge:

$$\sum_{i} x_{iO} = 1 \quad \text{for} \quad i \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\} \tag{6}$$

Ensure that the starting city has exactly one outgoing edge:

$$\sum_{j} x_{Aj} = 0 \quad \text{for} \quad j \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\} \tag{7}$$

$x_{ij}, y_i \in \{0, 1\}, \quad \forall i, j \in \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$

These constraints ensure that:

- There is exactly one path from the starting city to the destination city.
- There are no cycles in the path.
- Each city (excluding the starting and destination cities) may not necessarily be visited exactly once.

Solving this linear programming problem will give us the shortest path between the given cities both in terms of distances and likelikood of delays.

4 Receiving and reading the data

[131]:
```python
# Obtaining given data
import numpy as np
from numpy import genfromtxt


# cities = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',␣
↪'N', 'O'][:5]
cities = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',␣
↪'O']
start_city = cities[0]
destination_city = cities[-1]

# Number of cities
num_cities = len(cities)

# my_distances = genfromtxt('distance.csv', delimiter=',')[:5, :5]
my_distances = genfromtxt('distance.csv', delimiter=',')
my_delays = genfromtxt('delay.csv', delimiter=',')
# my_distances[1, 3] = 6
# my_distances[3, 1] = 6

# distances = my_distances[:4, :4]
d = my_distances.flatten()
rhs_binary = np.zeros(num_cities-2)
d = np.concatenate((d, rhs_binary))

p = my_delays.flatten()
rhs_binary = np.zeros(num_cities-2)
p = np.concatenate((p, rhs_binary))

# print(my_distances)
# print(my_delays)
```

5 Creating the constraints

[132]:
```
def get_constraints(file_name: str):

    """
    Generate constraints for an optimization problem based on given data.

    Args:
    - file_name (str): The name of the file containing the data.
                       It should be either 'distance.csv' or 'delay.csv'.

    Returns:
    - distances (numpy.ndarray): The distances between cities.
    - A_ineq (None): Inequality constraint matrix (None for this function).
    - b_ineq (None): Inequality bounds (None for this function).
    - Ae (numpy.ndarray): Equality constraint matrix.
    - be (numpy.ndarray): Equality bounds.
    """

    if file_name == 'distance.csv':
        distances = my_distances
    elif file_name == 'delay.csv':
        distances = my_delays
    else:
        print("File not found")


    # Distance between cities (replace with actual distances)
    # distances = my_distances[:4, :4]
    # distances = my_distances
    # delays = my_delays

    # Find the maximum distance
    max_distance = np.max(distances)

    # Update main diagonal elements
    np.fill_diagonal(distances, max_distance + 1)
    # print("distances:")
    # print(distances)

    # Inequality Constraints:

    # Constraint coefficients for exactly zero or one outgoing edge from each␣
 ↪inner city(except for A, O)
    A_outgoing = []
    b_outgoing = []
    for i_index, i in enumerate(cities[1:-1]):  # Exclude the last city
```

```
        # print(i_index, i)
        row = np.zeros(num_cities ** 2)
        rhs_binary = np.zeros(num_cities-2) # to count # of outgoing edges from each inner cities, B, C, ..., N.
        rhs_binary[i_index] = -1 # outgoing = incoming, so same rhs binaries
        for j in cities:
            if j != i and distances[cities.index(i), cities.index(j)] > 0:
                j_index = cities.index(j)
                row[num_cities * cities.index(i) + cities.index(j)] = 1
        row = np.concatenate((row, rhs_binary))
        A_outgoing.append(row)
        b_outgoing.append(0)
    # print("A_outgoing:")
    # print(np.array(A_outgoing))

    # Constraint coefficients for exactly zero or one incoming edge from each inner city(except for A, O)
    A_incoming = []
    b_incoming = []

    for j_index, j in enumerate(cities[1:-1]):  # Exclude the last city
        row = np.zeros(num_cities ** 2)
        rhs_binary = np.zeros(num_cities-2) # to count # of incoming edges from each inner cities, B, C, ..., N.
        rhs_binary[j_index] = -1 # outgoing = incoming, so same rhs binaries
        for i_index, i in enumerate(cities):
            if i != j and distances[cities.index(i), cities.index(j)] > 0:
                row[num_cities * cities.index(i) + cities.index(j)] = 1
        row = np.concatenate((row, rhs_binary))
        A_incoming.append(row)
        b_incoming.append(0)
    # print("A_incoming:")
    # print(np.array(A_incoming))

    A = np.vstack((A_outgoing, A_incoming))
    b = np.concatenate((b_outgoing, b_incoming))
    # print("\nInequality Constraints (A):")
    # print(np.array(A))
    # print("\nInquality Bounds (b):")
    # print(np.array(b))

    # Equality Constraints
    A_eq = []
    b_eq = []

    # Destination city constraint for having 0 outgoing edges
    row = np.zeros(num_cities ** 2)
```

```
    rhs_binary = np.zeros(num_cities-2)
    for j in cities:
        if distances[cities.index(destination_city), cities.index(j)] > 0:
            row[num_cities * cities.index(destination_city) + cities.index(j)] = 1
    row = np.concatenate((row, rhs_binary))
    A_eq.append(row)
    b_eq.append(0)
    # print("A_eq:")
    # print(np.array(A_eq))


    # Starting city constraint for having zero incoming edges
    row = np.zeros(num_cities ** 2)
    rhs_binary = np.zeros(num_cities-2)
    for i in cities:
        if distances[cities.index(i), cities.index(start_city)] > 0:
            row[num_cities * cities.index(i) + cities.index(start_city)] = 1
    row = np.concatenate((row, rhs_binary))
    A_eq.append(row)
    b_eq.append(0)
    # print("A_eq:")
    # print(np.array(A_eq))

    # Destination city constraint for having 1 incoming edge
    row = np.zeros(num_cities ** 2)
    rhs_binary = np.zeros(num_cities-2)
    for i in cities:
        if distances[cities.index(i), cities.index(destination_city)] > 0:
            row[num_cities * cities.index(i) + cities.index(destination_city)] = 1
    row = np.concatenate((row, rhs_binary))
    A_eq.append(row)
    b_eq.append(1)
    # print("A_eq:")
    # print(np.array(A_eq))

    # Starting city constraint for having 1 outgoing edge
    row = np.zeros(num_cities ** 2)
    rhs_binary = np.zeros(num_cities-2)
    for j in cities:
        if distances[cities.index(start_city), cities.index(j)] > 0:
            row[num_cities * cities.index(start_city) + cities.index(j)] = 1
    row = np.concatenate((row, rhs_binary))
    A_eq.append(row)
    b_eq.append(1)
    # print("A_eq:")
```

```python
    # print(np.array(A_eq))

    Ae = np.vstack((A, A_eq)) # stacking all LHS of constraints together
    be = np.concatenate((b, b_eq)) # stacking all RHS of constraints together
    A_ineq = None
    b_ineq = None


    return distances, A_ineq, b_ineq, Ae, be

if __name__ == "__main__":

    distances_received, A_ineq, b_ineq, Ae, be = get_constraints('distance.csv')
    print("distances_received :")
    print(distances_received)

    print("\nEquality Constraints (Ae):")
    print(np.array(Ae))
    print("\nEquality Bounds (be):")
    print(np.array(be))
    print("Ae.shape: ", np.array(Ae).shape)
```

```
distances_received :
[[74.  0.  0. 42.  0.  0.  0. 59. 29.  0.  0. 25.  0.  0.  0.]
 [ 0. 74.  0.  8. 21.  0.  0.  0.  0.  0. 25.  0.  0.  0. 25.]
 [ 0.  0. 74.  0.  0.  0. 31. 62.  0. 11.  0.  0. 40.  0.  0.]
 [42.  8.  0. 74. 28.  0.  0.  0. 42.  0. 29. 40.  0.  0.  0.]
 [ 0. 21.  0. 28. 74.  0.  0.  0.  0.  0.  0. 62. 65.  0. 26.]
 [ 0.  0.  0.  0.  0. 74. 25. 10. 28.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 31.  0.  0. 25. 74. 32. 41.  0. 50.  0. 54.  0.  0.]
 [59.  0. 62.  0.  0. 10. 32. 74. 35. 73.  0.  0.  0.  0.  0.]
 [29.  0.  0. 42.  0. 28. 41. 35. 74.  0. 46.  0.  0.  0.  0.]
 [ 0.  0. 11.  0.  0.  0.  0. 73.  0. 74.  0.  0. 40.  0.  0.]
 [ 0. 25.  0. 29.  0.  0. 50.  0. 46.  0. 74.  0. 40.  9. 13.]
 [25.  0.  0. 40. 62.  0.  0.  0.  0.  0.  0. 74.  0.  0.  0.]
 [ 0.  0. 40.  0. 65.  0. 54.  0.  0. 40. 40.  0. 74. 33. 40.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  9.  0. 33. 74.  9.]
 [ 0. 25.  0.  0. 26.  0.  0.  0.  0.  0. 13.  0. 40.  9. 74.]]

Equality Constraints (Ae):
[[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [1. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [1. 0. 0. … 0. 0. 0.]]
```

```
Equality Bounds (be):
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
Ae.shape:  (30, 238)
```

6 SOLVING THE INTEGER PROBLEM PROBLEM

[133]:
```
import pandas as pd
import numpy as np
import scipy.optimize as opt
from os import strerror

"""
All decision variables Xij, Yi are binaries from {0, 1}:
"""



def solve_IP(file_name: str, objective_vector):

    """
    Solves an Integer Programming (IP) problem given the data file and the␣
 ↪objective vector.

    Args:
    - file_name (str): The name of the file containing the data.
    - objective_vector (numpy.ndarray): The objective vector for the IP problem.

    Returns:
    - objecutive_value (float): The optimal objective value.
    - solution (numpy.ndarray): The optimal solution vector.
    """

    # The problem we will solve is:

    # min z  =  dAA*xAA +  dAB*xAB + ... + dOA*xOA + ...+ dOO*xOO + 0*yB + 0*yC␣
 ↪+ ... + 0*yN

    # s.t.    Ae x = be

    #         xAA, xAB, ... ,  xOO, yB, yC, ..., yN in {0, 1}


    # First build the objective vector ( matrix dij of distances from city i to␣
 ↪j):
    c = objective_vector
    distances, A_ineq, b_ineq, Ae, be = get_constraints(file_name = file_name)
```

```python
    # Next, create the coefficient array for the inequality constraints.
    # Note that the inequalities must be Ax <= b, so some sign
    # changes result when converting >= into <=.
    # A_ineq = None # if no inequality constraints


    # Next the right-hand-side vector for the inequalities
    # Sign changes can occur here too.
    # b_ineq = None # if no inequality constraints



    #The coefficient matrix for the equality constraints and
    # the right hand side vector.
    # Ae = None # if no equality constraints

    # be = None # if no equality constraints

    # Next, we provide any lower and upper bound vectors, one
    # value for each decision variable.  In this example all
    # lower bound are zero and there are no upper bounds.
    bounds = [(0, 1) for _ in range(num_cities**2 + num_cities - 2)] #␣
↪considering added rhs_binary variables
    # print("bounds.shape = ", bounds.shape )

    # Lastly, we can specify which variables are required to be integer.
    # If no variables are integer then isint=[];  In our example, only x2 is␣
↪integer.
    # isint = np.ones(num_cities + num_cities - 2)
    isint = [1 for _ in range(num_cities**2 + num_cities - 2)]
    # print("isint.shape = ", isint.shape)

    # The call to the mixed integer solver looks like the following.
    # Notice that we pass usual "c" when we have a minimization
    # problem, we send "-c" when we have max problem.
    # This is because the solver is expecting a minimization.

    res=opt.linprog(c, A_ineq, b_ineq, Ae, be, bounds, integrality = isint)

    # The result is stored in the dictionary variable "res".
    # In particular, to show the optimal objective value and the
    # optimal decision variable values:

    objecvtive_value = res['fun']
    solution = np.array(res['x'])
```

```
    return objecvtive_value, solution

if __name__ == "__main__":

    objecvtive_value, solution = solve_IP(file_name = 'distance.csv',␣
 ↪objective_vector = d)
    print("objective vector :")
    print(d)
    print("\nmin z = ", objecvtive_value)
    print("\nat optimal solution x :")
    print(solution)
```

```
objective vector :
[ 0.  0.  0. 42.  0.  0.  0. 59. 29.  0.  0. 25.  0.  0.  0.  0.  0.  0.
  8. 21.  0.  0.  0.  0.  0. 25.  0.  0.  0. 25.  0.  0.  0.  0.  0.  0.
 31. 62.  0. 11.  0.  0. 40.  0.  0. 42.  8.  0.  0. 28.  0.  0.  0. 42.
  0. 29. 40.  0.  0.  0.  0. 21.  0. 28.  0.  0.  0.  0.  0.  0.  0. 62.
 65.  0. 26.  0.  0.  0.  0.  0.  0. 25. 10. 28.  0.  0.  0.  0.  0.  0.
  0.  0. 31.  0.  0. 25.  0. 32. 41.  0. 50.  0. 54.  0.  0. 59.  0. 62.
  0.  0. 10. 32.  0. 35. 73.  0.  0.  0.  0.  0. 29.  0.  0. 42.  0. 28.
 41. 35.  0.  0. 46.  0.  0.  0.  0.  0.  0. 11.  0.  0.  0.  0. 73.  0.
  0.  0.  0. 40.  0.  0.  0. 25.  0. 29.  0.  0. 50.  0. 46.  0.  0.  0.
 40.  9. 13. 25.  0.  0. 40. 62.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0. 40.  0. 65.  0. 54.  0.  0. 40. 40.  0.  0. 33. 40.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  9.  0. 33.  0.  9.  0. 25.  0.  0. 26.  0.
  0.  0.  0.  0. 13.  0. 40.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.]

min z =  75.0

at optimal solution x :
[ 0.  0.  0.  1.  0.  0.  0.  0. -0.  0.  0. -0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0. -0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  1.  0.  0.  0.  0. -0.  0.
 -0. -0.  0. -0.]
```

7 PRESENTING SOLUTIONS

[149]:
```
import pandas as pd
import numpy as np
import scipy.optimize as opt
from os import strerror

# Define lists to store the data
alfa_list = []
beta_list = []
optimal_path_list = []
cost_of_optimal_path_list = []
total_distance_list = []
delay_likelihood_list = []

def call_LP_solver(alfa: float, beta:float, file_name: str):
    """
      Solves the integer programming problem with the given alfa and beta␣
 ↪values.

      Args:
      - alfa (float): Weighting factor for distance in the objective function.
      - beta (float): Weighting factor for delay in the objective function.
      - file_name (str): Name of the file containing distance and delay data.

      Returns:
      None

      """

    global alfa_list, beta_list, optimal_path_list, cost_of_optimal_path_list,␣
 ↪total_distance_list, delay_likelihood_list
    # call_LP_solver does  c = alfa*distance + beta*delay as an objective␣
 ↪vector to solve IP
    average_d = d/np.mean(d)
    average_p = p/np.mean(p)
    c = alfa*average_d + beta*average_p
    # c = alfa*d + beta*p
    objecvtive_value, solution = solve_IP(file_name = file_name,␣
 ↪objective_vector = c)
    # print(objecutive_value)
    # print(solution)
    # print("Optimal path in binary matrix form:")
    # print(solution[:num_cities**2].reshape(num_cities, num_cities))

    # Store the data in lists
    alfa_list.append(alfa)
```

```
    beta_list.append(beta)

    # Execute the code
    if any(solution):
        # print("Optimal path:")
        path = [start_city]  # Start from city 'A'
        path_distance = []
        next_city_index = 0  # Start from the first city in the solution vector
        while path[-1] != destination_city:  # Continue until we reach the␣
↪destination city 'E'
            # Find the index of the next city with a value of 1 in the solution␣
↪vector
            solution_slice = solution[next_city_index * len(cities):␣
↪(next_city_index + 1) * len(cities)]
            # destination_slice = d[next_city_index * len(cities):␣
↪(next_city_index + 1) * len(cities)]
            destination_slice = alfa*d[next_city_index * len(cities):␣
↪(next_city_index + 1) * len(cities)] + beta*p[next_city_index * len(cities):␣
↪(next_city_index + 1) * len(cities)]
            next_city_index = np.where(solution_slice == 1)[0][0]

            # print(np.dot(solution_slice, destination_slice))
            dist = str(np.dot(solution_slice, destination_slice))
            path_distance.append(dist)

            # Convert the index to the corresponding city label
            next_city = cities[next_city_index]
            path.append(next_city)

        # Print the optimal path
        print(' --> '.join(path))
        print(' + '.join(path_distance), f' = {eval("+ ".join(path_distance))}')
        # Splitting into integer and fractional parts
        int_parts = [int(float(num)) for num in path_distance]
        frac_parts = [float(num) - int(float(num)) for num in path_distance]
        # Creating integ and frac parts

        integ = ' + '.join([f'{num:.6f}' for num in int_parts])
        frac = ' + '.join([f'{num:.6f}' for num in frac_parts])
        print(integ, f' = {eval(integ)}', ' - "total" distance')
        print(frac, f' = {eval(frac)}', ' - delay likelihood')

        optimal_path_list.append(' --> '.join(path))
        # cost_of_optimal_path_list.append(' + '.join(map(str, path_distance)))
        # cost_of_optimal_path_list.append(' + '.join(path_distance) + f' =␣
↪{eval("+ ".join(path_distance))}')
```

```
        cost_of_optimal_path_list.append(f'{eval("+ ".join(path_distance))}'[:
↪4])
        # total_distance_list.append(integ + f' = {eval(integ)}')
        total_distance_list.append(f'{eval(integ)}'[:4])
        # delay_likelihood_list.append(frac + f' = {eval(frac)}')
        delay_likelihood_list.append(f'{eval(frac)}'[:9])
    else:
        # print("No solution found.")
        optimal_path_list.append("No solution found.")
        cost_of_optimal_path_list.append("No solution found.")
        total_distance_list.append("No solution found.")
        delay_likelihood_list.append("No solution found.")

if __name__ == "__main__":

    alfa = 1
    beta = 0
    file_name = 'distance.csv'
    call_LP_solver(alfa, beta, file_name)
```

```
A --> D --> B --> O
42.0 + 8.0 + 25.0  = 75.0
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000000 + 0.000000 + 0.000000  = 0.0  - delay likelihood
```

8 1. Printing the solution for shortest distance part by ignoring any possible delays

[150]:
```
call_LP_solver(alfa = 1, beta = 0, file_name = 'distance.csv')
```

```
A --> D --> B --> O
42.0 + 8.0 + 25.0  = 75.0
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000000 + 0.000000 + 0.000000  = 0.0  - delay likelihood
```

9 2. Printing the solution for minimum delay likelihood part by ignoring any possible long distances

[151]:
```
call_LP_solver(alfa = 0, beta = 1, file_name = 'delay.csv')
```

```
A --> D --> E --> O
0.005126 + 0.0017569 + 0.008132  = 0.015014900000000001
0.000000 + 0.000000 + 0.000000  = 0.0  - "total" distance
0.005126 + 0.001757 + 0.008132  = 0.015015  - delay likelihood
```

10 3. Printing the solution for minimizing the weighted sum of travel time and delay likelihood

[152]:
```
print("alfa*distances + beta*delays likelihood:")
alfa = float(input('alfa = '))
beta = float(input('beta = '))
file_name = 'distance.csv'
call_LP_solver(alfa = alfa, beta = beta, file_name = file_name)
```

```
alfa*distances + beta*delays likelihood:
alfa = 2
beta = 3
A --> D --> E --> O
84.015378 + 56.0052707 + 52.024396  = 192.04504469999998
84.000000 + 56.000000 + 52.000000  = 192.0  - "total" distance
0.015378 + 0.005271 + 0.024396  = 0.045045  - delay likelihood
```

[153]:
```
print("alfa*distances + beta*delays likelihood:")
alfa = float(input('alfa = '))
# beta = float(input('beta = '))
file_name = 'distance.csv'
for beta in np.arange(0, 1.1, 0.1):
  print(f"when alfa = {alfa}, beta = {beta} optimal path:")
  call_LP_solver(alfa = alfa, beta = beta, file_name = file_name)
  print()
```

```
alfa*distances + beta*delays likelihood:
alfa = 1
when alfa = 1.0, beta = 0.0 optimal path:
A --> D --> B --> O
42.0 + 8.0 + 25.0  = 75.0
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000000 + 0.000000 + 0.000000  = 0.0  - delay likelihood

when alfa = 1.0, beta = 0.1 optimal path:
A --> D --> B --> O
42.0005126 + 8.0017342 + 25.0030107  = 75.0052575
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000513 + 0.001734 + 0.003011  = 0.005258  - delay likelihood

when alfa = 1.0, beta = 0.2 optimal path:
A --> D --> B --> O
42.0010252 + 8.0034684 + 25.0060214  = 75.010515
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001025 + 0.003468 + 0.006021  = 0.010514  - delay likelihood

when alfa = 1.0, beta = 0.30000000000000004 optimal path:
```

```
A --> D --> E --> O
42.0015378 + 28.00052707 + 26.0024396  = 96.00450447
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.001538 + 0.000527 + 0.002440  = 0.004505  - delay likelihood

when alfa = 1.0, beta = 0.4 optimal path:
A --> D --> E --> O
42.0020504 + 28.00070276 + 26.0032528  = 96.00600596
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.002050 + 0.000703 + 0.003253  = 0.006006  - delay likelihood

when alfa = 1.0, beta = 0.5 optimal path:
A --> D --> E --> O
42.002563 + 28.00087845 + 26.004066  = 96.00750744999999
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.002563 + 0.000878 + 0.004066  = 0.007507  - delay likelihood

when alfa = 1.0, beta = 0.6000000000000001 optimal path:
A --> D --> E --> O
42.0030756 + 28.00105414 + 26.0048792  = 96.00900894
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.003076 + 0.001054 + 0.004879  = 0.009009  - delay likelihood

when alfa = 1.0, beta = 0.7000000000000001 optimal path:
A --> D --> E --> O
42.0035882 + 28.00122983 + 26.0056924  = 96.01051043
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.003588 + 0.001230 + 0.005692  = 0.01051  - delay likelihood

when alfa = 1.0, beta = 0.8 optimal path:
A --> D --> E --> O
42.0041008 + 28.00140552 + 26.0065056  = 96.01201191999999
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.004101 + 0.001406 + 0.006506  = 0.012013  - delay likelihood

when alfa = 1.0, beta = 0.9 optimal path:
A --> D --> E --> O
42.0046134 + 28.00158121 + 26.0073188  = 96.01351341
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.004613 + 0.001581 + 0.007319  = 0.013513  - delay likelihood

when alfa = 1.0, beta = 1.0 optimal path:
A --> D --> E --> O
42.005126 + 28.0017569 + 26.008132  = 96.0150149
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.005126 + 0.001757 + 0.008132  = 0.015015  - delay likelihood
```

[154]:
```
print("alfa*distances + beta*delays likelihood:")
alfa = float(input('alfa = '))
# beta = float(input('beta = '))
file_name = 'distance.csv'
for beta in np.arange(0.1, 0.3, 0.01):
  print(f"when alfa = {alfa}, beta = {beta} optimal path:")
  call_LP_solver(alfa = alfa, beta = beta, file_name = file_name)
  print()
```

```
alfa*distances + beta*delays likelihood:
alfa = 1
when alfa = 1.0, beta = 0.1 optimal path:
A --> D --> B --> O
42.0005126 + 8.0017342 + 25.0030107  = 75.0052575
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000513 + 0.001734 + 0.003011  = 0.005258  - delay likelihood

when alfa = 1.0, beta = 0.11 optimal path:
A --> D --> B --> O
42.00056386 + 8.00190762 + 25.00331177  = 75.00578325
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000564 + 0.001908 + 0.003312  = 0.005784  - delay likelihood

when alfa = 1.0, beta = 0.12 optimal path:
A --> D --> B --> O
42.00061512 + 8.00208104 + 25.00361284  = 75.006309
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000615 + 0.002081 + 0.003613  = 0.006309  - delay likelihood

when alfa = 1.0, beta = 0.13 optimal path:
A --> D --> B --> O
42.00066638 + 8.00225446 + 25.00391391  = 75.00683475
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000666 + 0.002254 + 0.003914  = 0.006834  - delay likelihood

when alfa = 1.0, beta = 0.13999999999999999 optimal path:
A --> D --> B --> O
42.00071764 + 8.00242788 + 25.00421498  = 75.0073605
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000718 + 0.002428 + 0.004215  = 0.0073609999999999995  - delay likelihood

when alfa = 1.0, beta = 0.14999999999999997 optimal path:
A --> D --> B --> O
42.0007689 + 8.0026013 + 25.00451605  = 75.00788625
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000769 + 0.002601 + 0.004516  = 0.007886  - delay likelihood
```

```
when alfa = 1.0, beta = 0.15999999999999998 optimal path:
A --> D --> B --> O
42.00082016 + 8.00277472 + 25.00481712  = 75.00841199999999
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000820 + 0.002775 + 0.004817  = 0.008412  - delay likelihood

when alfa = 1.0, beta = 0.16999999999999998 optimal path:
A --> D --> B --> O
42.00087142 + 8.00294814 + 25.00511819  = 75.00893775
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000871 + 0.002948 + 0.005118  = 0.008937  - delay likelihood

when alfa = 1.0, beta = 0.17999999999999997 optimal path:
A --> D --> B --> O
42.00092268 + 8.00312156 + 25.00541926  = 75.0094635
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000923 + 0.003122 + 0.005419  = 0.009464  - delay likelihood

when alfa = 1.0, beta = 0.18999999999999995 optimal path:
A --> D --> B --> O
42.00097394 + 8.00329498 + 25.00572033  = 75.00998925
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.000974 + 0.003295 + 0.005720  = 0.009989000000000001  - delay likelihood

when alfa = 1.0, beta = 0.19999999999999996 optimal path:
A --> D --> B --> O
42.0010252 + 8.0034684 + 25.0060214  = 75.010515
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001025 + 0.003468 + 0.006021  = 0.010514  - delay likelihood

when alfa = 1.0, beta = 0.20999999999999996 optimal path:
A --> D --> B --> O
42.00107646 + 8.00364182 + 25.00632247  = 75.01104075
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001076 + 0.003642 + 0.006322  = 0.011040000000000001  - delay likelihood

when alfa = 1.0, beta = 0.21999999999999995 optimal path:
A --> D --> B --> O
42.00112772 + 8.00381524 + 25.00662354  = 75.0115665
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001128 + 0.003815 + 0.006624  = 0.011567  - delay likelihood

when alfa = 1.0, beta = 0.22999999999999995 optimal path:
A --> D --> B --> O
42.00117898 + 8.00398866 + 25.00692461  = 75.01209225
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001179 + 0.003989 + 0.006925  = 0.012093  - delay likelihood
```

```
when alfa = 1.0, beta = 0.23999999999999994 optimal path:
A --> D --> B --> O
42.00123024 + 8.00416208 + 25.00722568  = 75.012618
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001230 + 0.004162 + 0.007226  = 0.012618  - delay likelihood

when alfa = 1.0, beta = 0.24999999999999992 optimal path:
A --> D --> B --> O
42.0012815 + 8.0043355 + 25.00752675  = 75.01314375
42.000000 + 8.000000 + 25.000000  = 75.0  - "total" distance
0.001281 + 0.004335 + 0.007527  = 0.013143  - delay likelihood

when alfa = 1.0, beta = 0.2599999999999999 optimal path:
A --> D --> E --> O
42.00133276 + 28.000456794 + 26.00211432  = 96.003903874
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.001333 + 0.000457 + 0.002114  = 0.003904  - delay likelihood

when alfa = 1.0, beta = 0.2699999999999999 optimal path:
A --> D --> E --> O
42.00138402 + 28.000474363 + 26.00219564  = 96.004054023
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.001384 + 0.000474 + 0.002196  = 0.004054  - delay likelihood

when alfa = 1.0, beta = 0.2799999999999999 optimal path:
A --> D --> E --> O
42.00143528 + 28.000491932 + 26.00227696  = 96.004204172
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.001435 + 0.000492 + 0.002277  = 0.004204  - delay likelihood

when alfa = 1.0, beta = 0.2899999999999999 optimal path:
A --> D --> E --> O
42.00148654 + 28.000509501 + 26.00235828  = 96.004354321
42.000000 + 28.000000 + 26.000000  = 96.0  - "total" distance
0.001487 + 0.000510 + 0.002358  = 0.0043549999999999995  - delay likelihood
```

[155]:
```python
def create_dataframe():
    # Create DataFrame from the lists
    df = pd.DataFrame({
        'alfa': alfa_list,
        'beta': beta_list,
        'optimal path': optimal_path_list,
        'cost of path': cost_of_optimal_path_list,
        'distance': total_distance_list,
        'delay': delay_likelihood_list
    })
```

```python
    return df

if __name__ == "__main__":
    result_df = create_dataframe()
    print(result_df)
```

```
    alfa  beta         optimal path cost of path distance      delay
0    1.0  0.00  A --> D --> B --> O         75.0      75.0        0.0
1    1.0  0.00  A --> D --> B --> O         75.0      75.0        0.0
2    0.0  1.00  A --> D --> E --> O         0.01       0.0   0.015015
3    2.0  3.00  A --> D --> E --> O         192.      192.   0.045045
4    1.0  0.00  A --> D --> B --> O         75.0      75.0        0.0
5    1.0  0.10  A --> D --> B --> O         75.0      75.0   0.005258
6    1.0  0.20  A --> D --> B --> O         75.0      75.0   0.010514
7    1.0  0.30  A --> D --> E --> O         96.0      96.0   0.004505
8    1.0  0.40  A --> D --> E --> O         96.0      96.0   0.006006
9    1.0  0.50  A --> D --> E --> O         96.0      96.0   0.007507
10   1.0  0.60  A --> D --> E --> O         96.0      96.0   0.009009
11   1.0  0.70  A --> D --> E --> O         96.0      96.0    0.01051
12   1.0  0.80  A --> D --> E --> O         96.0      96.0   0.012013
13   1.0  0.90  A --> D --> E --> O         96.0      96.0   0.013513
14   1.0  1.00  A --> D --> E --> O         96.0      96.0   0.015015
15   1.0  0.10  A --> D --> B --> O         75.0      75.0   0.005258
16   1.0  0.11  A --> D --> B --> O         75.0      75.0   0.005784
17   1.0  0.12  A --> D --> B --> O         75.0      75.0   0.006309
18   1.0  0.13  A --> D --> B --> O         75.0      75.0   0.006834
19   1.0  0.14  A --> D --> B --> O         75.0      75.0  0.0073609
20   1.0  0.15  A --> D --> B --> O         75.0      75.0   0.007886
21   1.0  0.16  A --> D --> B --> O         75.0      75.0   0.008412
22   1.0  0.17  A --> D --> B --> O         75.0      75.0   0.008937
23   1.0  0.18  A --> D --> B --> O         75.0      75.0   0.009464
24   1.0  0.19  A --> D --> B --> O         75.0      75.0  0.0099890
25   1.0  0.20  A --> D --> B --> O         75.0      75.0   0.010514
26   1.0  0.21  A --> D --> B --> O         75.0      75.0  0.0110400
27   1.0  0.22  A --> D --> B --> O         75.0      75.0   0.011567
28   1.0  0.23  A --> D --> B --> O         75.0      75.0   0.012093
29   1.0  0.24  A --> D --> B --> O         75.0      75.0   0.012618
30   1.0  0.25  A --> D --> B --> O         75.0      75.0   0.013143
31   1.0  0.26  A --> D --> E --> O         96.0      96.0   0.003904
32   1.0  0.27  A --> D --> E --> O         96.0      96.0   0.004054
33   1.0  0.28  A --> D --> E --> O         96.0      96.0   0.004204
34   1.0  0.29  A --> D --> E --> O         96.0      96.0  0.0043549
```

11 4. Printing the solution for the combined problem using weight selection with starting city A and ending location at city O.

Report about out findings and recommendations to the company:

In conclusion, our experiments reveal that in real-life scenarios, the distances between cities remain constant, indicating that alfa is always equal to 1. By running a loop for beta within the range of [0, 1], we observe a transition in the optimal path from "A → D → B → O" to "A → D → E → O" as beta increases from 0.1 to 0.3. To gain deeper insights into the range of beta values where the optimal path changes, we conduct a secondary loop within the range of [0.1, 0.3] with a step size of 0.01. Our research indicates that this transition occurs between beta = 0.25 and beta = 0.26. At beta = 0.25, the optimal path is "A → D → B → O" with a total distance of 75.0 miles and a delay likelihood of 0.013143. However, at beta = 0.26, the optimal path shifts to "A → D → E → O" with a total distance of 96.0 miles and a delay likelihood of 0.003904.

Ultimately, the choice between these two options rests on the company's resources. Opting for a shorter travel distance of 75 miles may result in a higher delay likelihood of 0.013143 (meaning 1.3 % chance of having delay), whereas selecting the longer route of 96 miles offers a lower delay likelihood of 0.003904 (meaning 0.3 % chance of having delay). The decision should be made considering the company's financial resources and the trade-off between travel distance and delay likelihood.

[81]:
```
# To download:
# !sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
# !jupyter nbconvert --to pdf /content/LS_Linear_Optimzation.ipynb
```

[81]:
```

```