Project Report

Project: Undirected Multiple TSP Problem

Undirected Multiple Traveling Salesman Problem (mTSP) with Single Depot

Objective: Find the optimal set of tours for multiple salespersons starting from a single depot to cover a set of cities, minimizing the total travel cost or distance.

Undirected Graph: The graph is undirected, meaning the travel cost is the same in both directions between any two cities.

Depot: A single starting point for all 4 salespersons.

Salespersons: 4 salespersons or vehicles, each of whom must return to the depot after completing their tour. Cities: A set of cities that must be visited. Each city needs to be visited at least once by one of the salespersons.

Coverage: Each city must be visited exactly once, either by one or more salespersons.
Tour Limits: Depending on the problem, there might be constraints on the maximum number of cities a salesperson can visit or the maximum tour length.
Depot Return: Each salesperson must return to the depot after completing their tour.

Computational Complexity:

- **NP-Hard**: The mTSP is an NP-hard problem, meaning it is computationally challenging to find exact solutions for large instances.
- **Scalability**: The complexity grows with the number of salespersons and cities, requiring efficient algorithms for large-scale problems.

Evaluation Metrics:

- **Total Distance**: The sum of distances traveled by all salespersons.
- **Tour Length**: The length of each individual tour.
- **Computational Time**: Time taken to compute the solution.

```
!pip install pulp
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting pulp
  Downloading PuLP-2.9.0-py3-none-any.whl.metadata (5.4 kB)
Downloading PuLP-2.9.0-py3-none-any.whl (17.7 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.7/17.7 MB 16.6 MB/s eta 0:00:0000:0100:01
Installing collected packages: pulp
  WARNING: The script pulptest is installed in '/home/23m1505/.local/bin' which is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pulp-2.9.0
```

```
from pulp import *
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sn

#a handful of sites
#sites = ['Barcelona','Belgrade','Berlin','Brussels','Bucharest','Budapest','Copenhagen','Dubli
# sites = ['Barcelona','Belgrade','Berlin','Brussels','Bucharest','Budapest','Copenhagen']
latlng = ['latitude', 'longitude']
posit = pd.read_csv('india_city.csv', index_col=None)
#flighttime = pd.read_csv('flight_time.csv', index_col="City")
#distance = pd.read_csv('distance.csv', index_col="City")
#print(len(position["city"]))
position=posit.head(20)
position
```

	city	lat	lng
0	Delhi	28.6100	77.2300
1	Mumbai	19.0761	72.8775
2	Kolkāta	22.5675	88.3700
3	Bangalore	12.9789	77.5917
4	Chennai	13.0825	80.2750
5	Hyderābād	17.3617	78.4747
6	Pune	18.5203	73.8567
7	Ahmedabad	23.0225	72.5714
8	Sūrat	21.1702	72.8311
9	Lucknow	26.8500	80.9500
10	Jaipur	26.9000	75.8000
11	Kanpur	26.4499	80.3319
12	Mirzāpur	25.1460	82.5690
13	Nāgpur	21.1497	79.0806
14	Ghāziābād	28.6700	77.4200
15	Supaul	26.1260	86.6050
16	Vadodara	22.3000	73.2000
17	Rājkot	22.3000	70.7833
18	Vishākhapatnam	17.7042	83.2978
19	Indore	22.7167	75.8472

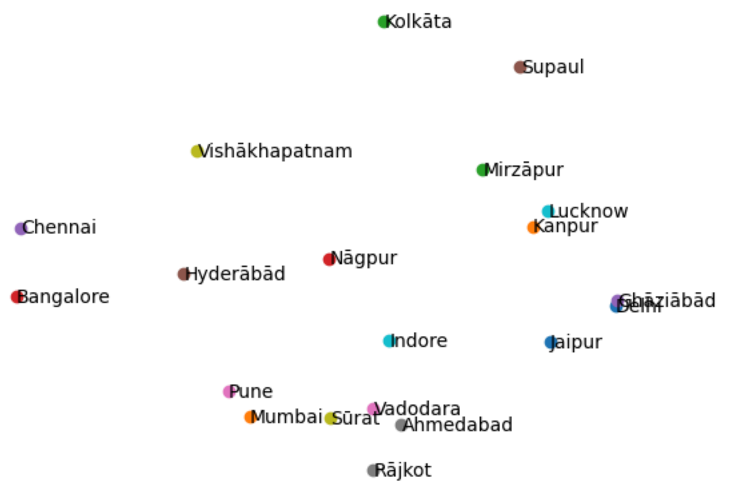
```
site= position["city"]
sites=[]
latitude=[]
longitude=[]
for i in range(len(position["city"])):
    sites.append(str(site[i]))
    latitude.append(position["lat"][i])
    longitude.append(position["lng"][i])
longitude[0]
```

```
77.23
```

```
#make some positions (so we can plot this)
positions = dict( ( sites[i], (latitude[i],longitude[i]) ) for i in range(len(position["city"])


for s in positions:
    p = positions[s]
    plt.plot(p[0],p[1],'o')
    plt.text(p[0]+.01,p[1],s,horizontalalignment='left',verticalalignment='center')

plt.gca().axis('off');
```



```
import math
import numpy as np

# Define the Haversine function
def haversine(lat1, lon1, lat2, lon2):
    """Calculate the great-circle distance between two points."""
    lat1, lon1, lat2, lon2 = map(math.radians, [lat1, lon1, lat2, lon2])

    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = math.sin(dlat / 2) ** 2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon / 2) ** 2
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    R = 6371.0  # Radius of Earth in kilometers
    distance = R * c
    return distance

# Calculate the distance matrix
def calculate_distance_matrix(coords):
    """Calculate the distance matrix for a list of coordinates."""
    num_cities = len(coords)
    distance_matrix = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            lat1, lon1 = coords[i]
            lat2, lon2 = coords[j]
            distance = haversine(lat1, lon1, lat2, lon2)
            distance_matrix[i, j] = distance
            distance_matrix[j, i] = distance  # Distance is symmetric

    return distance_matrix

# Example data (replace with actual data)
#sites = [f'City{i}' for i in range(len(position["city"]))]
longitude = [i for i in range(len(position["city"]))]  # Example data
latitude = [i for i in range(len(position["city"]))]   # Example data
#positions_ = dict((sites[i], (longitude[i], latitude[i])) for i in range(len(position["city"])

# Extract coordinates
coords = [positions[city] for city in sites]

# Calculate the distance matrix
distance_matrix = calculate_distance_matrix(coords)
distance_matrix
```

```
                905.94846341,  425.97917286,  619.63789511,  702.17326104,
                648.02124602,  661.64312547,  720.8991992 ,  602.93295259,
                569.72521057,    0.        ,  852.78538398,  944.96173264,
                620.71813499,  866.4549944 ,  585.03170511,  376.26827118],
              [  19.70587084, 1162.14356311, 1289.64690174, 1744.8611472 ,
               1758.22602031, 1262.02696806, 1185.33357645,  793.36351936,
                953.45327885,  401.96390563,  253.23866   ,  378.57019913,
                643.4538665 ,  852.78538398,    0.        ,  949.53978841,
                825.10794364,  971.91333692, 1358.66812049,  680.44947592],
              [ 965.7637594 , 1610.94687274,  434.19187859, 1738.81504426,
               1593.91060732, 1285.56606121, 1558.82401074, 1459.60539847,
               1506.16626491,  568.47726055, 1078.2150068 ,  626.37088621,
                418.99930373,  944.96173264,  949.53978841,    0.        ,
               1423.56457341, 1658.66731493,  996.51433877, 1152.7700674 ],
              [ 809.76569682,  360.0470858 , 1558.78487899, 1135.86430755,
               1269.09428836,  778.23794114,  425.81622837,  103.02687032,
                131.27913453,  932.45409666,  575.03480243,  856.87264423,
               1004.57596442,  620.71813499,  825.10794364, 1423.56457341,
                  0.        ,  248.62412316, 1171.83823497,  275.84723655],
              [ 954.16975311,  419.46205425, 1806.75920271, 1262.22177698,
               1434.74646735,  973.71484018,  528.36475659,  200.29306947,
                246.00575591, 1145.22579418,  720.1580651 , 1071.08148399,
               1240.28692927,  866.4549944 ,  971.91333692, 1658.66731493,
                248.62412316,    0.        , 1403.18065407,  522.21974119],
              [1361.44782767, 1109.85826376,  756.6910339 ,  806.34427205,
                607.49296878,  512.79111222, 1001.78202869, 1264.25202301,
               1162.83775835, 1045.17618098, 1280.03776821, 1019.23368164,
                830.91495387,  585.03170511, 1358.66812049,  996.51433877,
               1171.83823497, 1403.18065407,    0.        ,  956.2714531 ],
              [ 669.77889127,  508.9178722 , 1284.87660893, 1098.37527864,
               1168.9511934 ,  655.60535665,  510.50811288,  337.32975621,
                355.42793161,  690.2107406 ,  465.1861143 ,  614.67169828,
                734.50440964,  376.26827118,  680.44947592, 1152.7700674 ,
                275.84723655,  522.21974119,  956.2714531 ,    0.        ]])
```

```
# get distanc between cities
distances = dict( ((sites[i],sites[j]), distance_matrix[i][j] ) for i in range(len(position["ci
```

The model

With a few modifications, the original traveling salesman problem can support multiple salesman. Instead of making each facility only be visited once, the origin facility will be visited multiple times. If we have two salesman then the origin is visited exactly twice and so on.

For **K** vehicles or sales people:

Variables:

$$x_{ij} = \begin{cases} 1 : \text{the path goes from city i to j} \\ 0 : \text{otherwise} \end{cases}$$

order dummy variables:

$$u_i - u_j + C * x_{ij} \le C - d_j, C = N/K \begin{cases} u_i : \text{order that site i is visited} \\ d_j : \text{the cost to visit city j}, 0 \le u_i \le C - d_j, \forall i \in V \setminus \{0\} \end{cases}$$

Goal:

$$min \sum_{i=0}^{n} \sum_{j \ne i, j=0}^{n} c_{ij} x_{ij} \begin{cases} c_{ij} : \text{distance from city i to city j} \\ x_{ij} : \text{whether there's a path between i and j} \end{cases}$$

Constraints:

$$\sum_{i \in V} x_{ij} = 1, \forall j \in V \setminus \{0\}$$
$$\sum_{j \in V} x_{ij} = 1, \forall i \in V \setminus \{0\}$$
$$\sum_{i \in V} x_{i0} = K$$
$$\sum_{j \in V} x_{0j} = K$$

```
K = 4 #the number of sales people

#create the problme
prob=LpProblem("vehicle", LpMinimize)
```

```
#indicator variable if site i is connected to site j in the tour
x = LpVariable.dicts('x',distances, 0,1,LpBinary)
#dummy vars to eliminate subtours
u = LpVariable.dicts('u', sites, 0, len(sites)-1, LpInteger)

#the objective
cost = lpSum([x[(i,j)]*distances[(i,j)] for (i,j) in distances])
prob+=cost

#constraints
for k in sites:
    cap = 1 if k != 'Mumbai' else K
    #inbound connection
    prob+= lpSum([ x[(i,k)] for i in sites if (i,k) in x]) ==cap
    #outbound connection
    prob+=lpSum([ x[(k,i)] for i in sites if (k,i) in x]) ==cap

#subtour elimination
N=len(sites)/K
for i in sites:
    for j in sites:
        if i != j and (i != 'Mumbai' and j!= 'Mumbai') and (i,j) in x:
            prob += u[i] - u[j] <= (N)*(1-x[(i,j)]) - 1
```

Solve it!

```
%time prob.solve()
#prob.solve(GLPK_CMD(options=['--simplex']))
print(LpStatus[prob.status])
```

```
Cbc0010I After 907000 nodes, 51392 on tree, 10497.7 best solution, best possible 9992.3814 (1796.85 seconds)
Cbc0010I After 908000 nodes, 50858 on tree, 10497.7 best solution, best possible 9992.3814 (1798.41 seconds)
Cbc0010I After 909000 nodes, 50706 on tree, 10497.7 best solution, best possible 9992.3814 (1800.25 seconds)
Cbc0010I After 910000 nodes, 50890 on tree, 10497.7 best solution, best possible 9996.6369 (1803.01 seconds)
Cbc0010I After 911000 nodes, 50953 on tree, 10497.7 best solution, best possible 9996.6369 (1805.13 seconds)
Cbc0010I After 912000 nodes, 51033 on tree, 10497.7 best solution, best possible 9996.6369 (1806.96 seconds)
Cbc0010I After 913000 nodes, 50803 on tree, 10497.7 best solution, best possible 9996.6369 (1808.10 seconds)
Cbc0010I After 914000 nodes, 51011 on tree, 10497.7 best solution, best possible 10001.459 (1810.44 seconds)
Cbc0010I After 915000 nodes, 51093 on tree, 10497.7 best solution, best possible 10001.459 (1812.70 seconds)
Cbc0010I After 916000 nodes, 51149 on tree, 10497.7 best solution, best possible 10001.459 (1814.90 seconds)
```

```python
non_zero_edges = [ e for e in x if value(x[e]) != 0 ]

def get_next_site(parent):
    '''helper function to get the next edge'''
    edges = [e for e in non_zero_edges if e[0]==parent]
    for e in edges:
        non_zero_edges.remove(e)
    return edges

tours = get_next_site('Mumbai')
tours = [ [e] for e in tours ]

for t in tours:
    while t[-1][1] !='Mumbai':
        t.append(get_next_site(t[-1][1])[-1])
```

The optimal route:

```python
for t in tours:
    print(' -> '.join([ a for a,b in t]+['Mumbai']))
```

```
Mumbai -> Bangalore -> Chennai -> Vishākhapatnam -> Hyderābād -> Pune -> Mumbai
Mumbai -> Kanpur -> Lucknow -> Ghāziābād -> Delhi -> Jaipur -> Mumbai
Mumbai -> Nāgpur -> Kolkāta -> Supaul -> Mirzāpur -> Indore -> Mumbai
Mumbai -> Rājkot -> Ahmedabad -> Vadodara -> Sūrat -> Mumbai
```

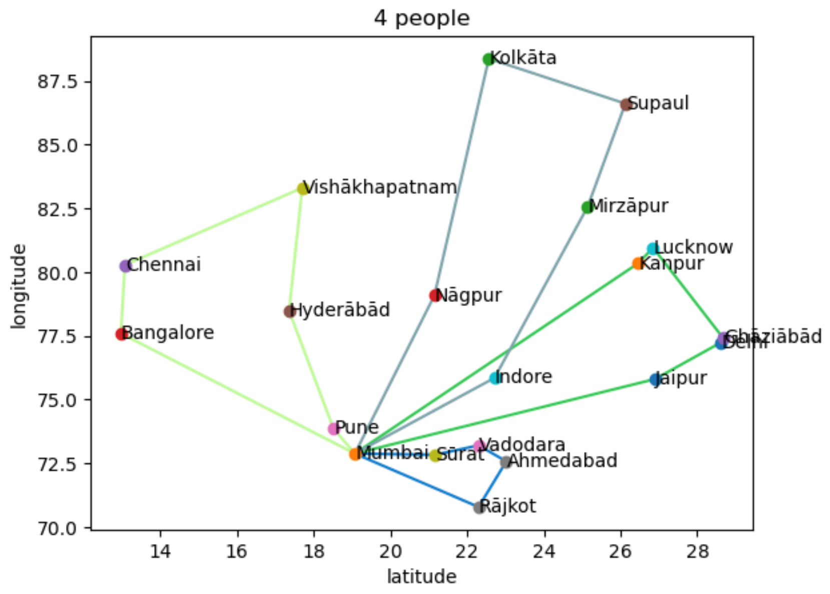
Calculate total time:

```python
#totalTime = 0;
#for t in tours:
#    time = 0
 #   for i in range(0, len(t)):
 #       time += flighttime.loc[t[i][0], t[i][1]]
#         print(flighttime.loc[t[i][0], t[i][1]])
#     print(time)
 #   if time > totalTime:
 #       totalTime = time
#print(totalTime)


#draw the tours
colors = [np.random.rand(3) for i in range(len(tours))]
for t,c in zip(tours,colors):
    for a,b in t:
        p1,p2 = positions[a], positions[b]
        plt.plot([p1[0],p2[0]],[p1[1],p2[1]], color=c)

#draw the map again
for s in positions:
    p = positions[s]
    plt.plot(p[0],p[1],'o')
    plt.text(p[0]+.01,p[1],s,horizontalalignment='left',verticalalignment='center')

plt.title('%d '%K + 'people' if K > 1 else 'person')
plt.xlabel('latitude')
plt.xlabel('latitude')
plt.ylabel('longitude')
# plt.gca().axis('off')
plt.show()
```



```
#print('Longest time spent:', totalTime, '(min)')
print('Total distance:', value(prob.objective), '(km)')
```

```
Total distance: 10497.700327091168 (km)
```