Problem Statement

The problem is to set up a connection between a mother and a daughter center. The mother and daughter are locations whose geo codes (latitudes & longitudes) are provided in the excel file (attached below). Each mother center has a set of vehicles which connects to the daughter centers. The connection should be such that a vehicle can leave the mother center not before 6 am and must reach the daughter center before 10 am. A vehicle can leave from 1 mother center and connect multiple daughter center, however the last daughter center that it connects should reach before 10 am. Each vehicle has a fixed base cost and additional cost based on time and distance travelled. The commercials of the vehicle are given below:

	Tata Ace
Base Kms	Base Rate
100	500
Rate per extra kms	10
Rate after 6 hours	50

You need to assign a daughter center to mother center by assuming that a vehicle to a daughter center will be connected by the mother center only.
You are free to make your assumptions, but you need to state them clearly.
The purpose is to understand your thought process in building a solution.
Submissions: Your submissions would include
i) A doc (max 2 page) explaining the methodology, solution, tools and language used. You can include the result summary and your code in the appendix (that can go beyond the 2 page)
ii) An Excel file with the name of mother center corresponding to each daughter center.

Solution

Deliverables :-

The deliverables of the current problem can be summarized into the following. :-

1. Assign the mother centers into daughter centers such that the total cost of travel is minimized.
2. Furthermore. the total travel time for each set of mother-daughter cluster is constrained to 4 hours.

Assumptions :-

1.All the vehicles in the network are assumed to be travelling at cruise speed i.e. 30 km/hr [Max speed along highways]. This assumption gives the optimization problem a homogeneous structure similar to that of a linear programming problem which can be solved with gradient descent under the problem constraints.

2. The routes from each depot are assumed to be starting and ending at that. same depot only.

2. A greedy approach is being used to solve the problem i.e. the problem always looks at finding the minimum cost path. There could be other considerations that we could have assumed but the greedy approach gives us a first pass at looking at the optimal structure of the solution.

Solution Methodology :-

Each vehicle has a fixed base cost and additional cost based on time and distance travelled. The commercials of the vehicle are given below:-

	Tata Ace
Base Kms	Base Rate
100	500
Rate per extra kms	10
Rate after 6 hours	50

Table 1.

From Table 1. and assuming that all the vehicles are traveling with the same cruise speed, we can identify that the cost metric is dependent on the number of vehicles and the total distance travelled.

The above problem can be considered to be a multi depot vehicle routing problem where the route plan schedule is the result of a multilevel optimization routine subject to the routing constraints. The problem is NP-complete and subject to specific constraints, the solution is feasible.

The structure of the solution to the problem is shown below:-

1.The higher level of the optimization problem selects the k-nearest neighbors (x_i, y_i) for each depot (x_{depot}, y_{depot}).[Greedy Approach].

$$\min \sum_{i=1}^{k} \left(\left(x_i - x_{depot}\right)^2 + \left(y_i - y_{depot}\right)^2 \right)^{0.5}$$

2.The lower level is a routing optimization problem for each mother center subject to time constraints.

Data:

c_{ij}: Cost of travel from i to j

Decision variable:

x_{ijk}: Travel direct from i to j on vehicle k

$$\text{minimise}: \sum_{i,j} \mathrm{c}_{\mathrm{ij}} \sum_{k} x_{ijk}$$

subject to

$$\begin{aligned}
\sum_{\mathrm{i}} \sum_{\mathrm{k}} \mathrm{x}_{\mathrm{ijk}} &= 1 \quad \forall j \\
\sum_{\mathrm{j}} \sum_{\mathrm{k}} \mathrm{x}_{\mathrm{ijk}} &= 1 \quad \forall i \\
\sum_{\mathrm{j}} \sum_{\mathrm{k}} \mathrm{x}_{\mathrm{ihk}} - \sum_{\mathrm{j}} \sum_{\mathrm{k}} \mathrm{x}_{\mathrm{hjk}} &= 0 \quad \forall k, h
\end{aligned}$$

$$\{\mathrm{x}_{\mathrm{ijk}}\} \subseteq S$$

$$x_{ijk} \in \{0,1\}$$

3.If any of the stations are violating the constraints inside a cluster, the number of trucks are increased from k =1 for the particular route in that cluster. If they still violate the constraints, the clusters are diluted and Steps 1,2 and 3 are continued till there are no stations left to visit.

Flowchart:-
A flowchart of the algorithm has been described using the following diagram below:-

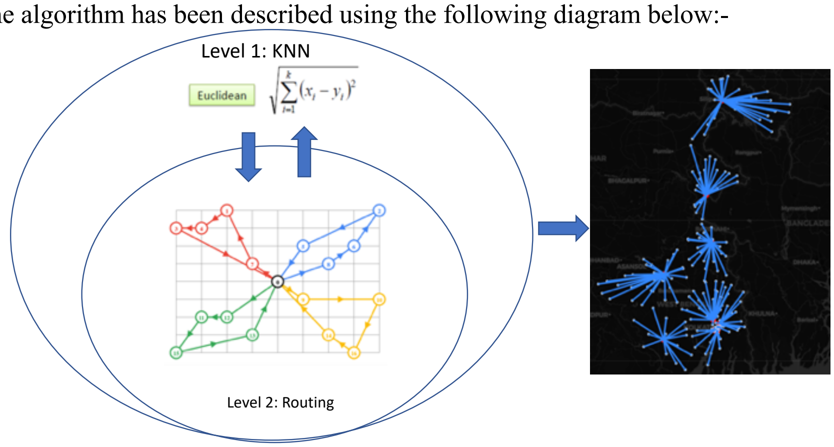


Figure 1.

Functions and tools used in the problem :-

Python3.6 has been used for solving the problem. The second level of the optimization problem has been solved with the help of **Google OR development kit** which consists of libraries specific to routing problems under various scenarios. Below has been described a high-level description of the. functions and how. they are being used.

Libraries Used:-
pandas = for dataframes.
sklearn = for calculating k-cluster centers based on minimum distance metric
folium,geopandas = for generation and visualization of data on maps
shapely= for fetching nearest points from spatial data on a map
ortools.constraint_solver = Google Operations Research Tool for solving the routing problem [pywrapcp, routing_enums_pb2 and Distance_Metric (for generating the distance matrix)]

Definitions and description of the functions:-

create_gdf :- The following function converts a dataframe into a GeoDataFrame for plotting maps from latitude and longitude values .The function also facilitates the calculation of nearest neighbours from each depot which has been described in the next definition.The function takes in all the points assigned as "Daughters" and all the points assigned as "Mothers" and converts them into two specific GeoDataFrames for later processing.

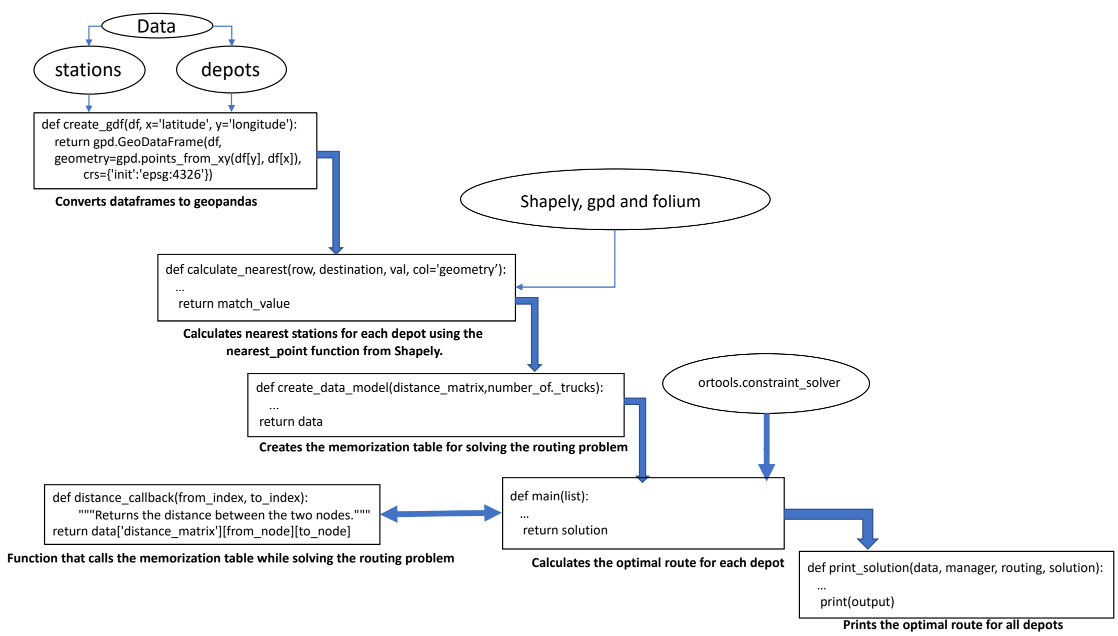
calculate_nearest :- This function calculates the points closest to a particular given point.The inputs to the function is a "Daughter" location and a GeoDataFrame of "Mother" locations.It uses the nearest_point function to calculate and compare the haversine distances from all "Mother" locations and assigns it to the nearest one.For our convenience,the "Mother" locations have been renamed serially as Mother0,Mother1 ….

create_data_model :- The data model block is simply a memorization table that holds the. distance matrix or the distances between all possible pairs of locations.The model reduces computational complexity for the routing component. The function is called once for each cluster.The function also denotes the starting row of the list as the "Mother" or the depot in each cluster (denoted as data['depot']=0). It also takes into account the number of vehicles in the network as an input.The vehicle number is initially kept at 1.If any of the constraints are violated, the number is increased by 1.

main:- The main function defines the objective for each arc in the route and calls pywrapcp and routing_enums_pb2 to solve the problem given added constraints on travel distance. The function traverses the list of depot centers and creates a data model for the routing problem using the **create_data_model** function.The distance matrix is created using sklearn's DistanceMetric.Since the data model and future processing involves coordinates in radian,conversion is done in line.After the creation of the data model, the routing model and the index managers are instantiated with the help of **pywrapcp**.

The constraints are added as a dimension to the problem.Since the vehicles are allowed to run at cruise speed of 30 km/hr,and the maximum allowed time for the entire trip is 4 hours,the maximum allowable distance translates to 120 km,which has been added as a constraint to the problem.

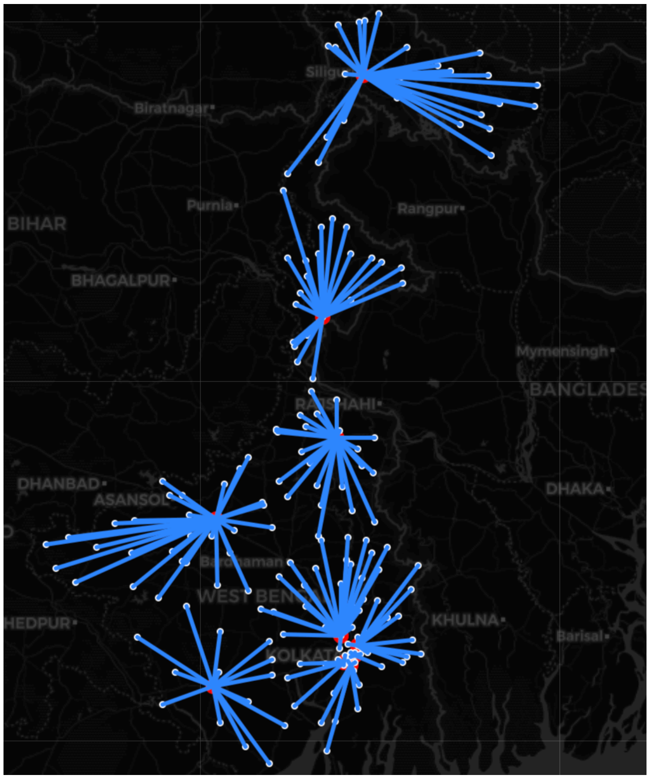
The problem is solved with the help of **SolveWithParameters function** that takes in search parameters based on PATH_CHEAPEST_ARC strategy decided by **routing_enums_pb2.Since the cost function here is completely focused on the distance mertric (assuming speed to be constant).**
The evaluation criteria **SetArcCostEvaluatorOfAllVehicles** calls the **distance_callback function** that looks at the memorization table to retrieve the distance values for cost calculation. Finally**,** the **print_solution** function prints the route and the cost calculated for each node.
The overall interaction between different components of the system has been shown below in the following chart.



Solution:-

The output to the problem is a map that shows all the clusters and the routes that minimize the total cost of travel. The cost is also displayed.The output route is displayed as indices (0-*n*) and *n* being the index corresponding to the *n*-th point in the cluster.In this case the solution converges at number of trucks being equal to 1.The assigned pairs have been saved in "out.csv" in the submission directory.

Clusters:



Optimal Route:-

Route for vehicle 0:
 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -
> 23 -> 0
Distance of the route: 23km

Cost of travel: 500INR
Maximum of the route distances: 23km
Route for vehicle 0:
 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -
> 23 -> 24 -> 25 -> 0
Distance of the route: 25km

Cost of travel: 500INR
Maximum of the route distances: 25km
Route for vehicle 0:
 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 0
Distance of the route: 16km

Cost of travel: 500INR
Maximum of the route distances: 16km

```
Route for vehicle 0:
 0 ->  26 ->  27 ->  28 ->  29 ->  30 ->  31 ->  32 ->  14 ->  15 ->  16 -
>  17 ->  18 ->  19 ->  20 ->  21 ->  22 ->  23 ->  24 ->  25 ->  1 ->  2
->  3 ->  4 ->  5 ->  6 ->  7 ->  8 ->  9 ->  10 ->  11 ->  12 ->  13 -> 0
Distance of the route: 100km

Cost of travel: 550INR
Maximum of the route distances: 100km
Route for vehicle 0:
 0 ->  1 ->  2 ->  3 ->  4 ->  5 ->  6 ->  7 ->  8 ->  9 ->  10 ->  11 ->
12 ->  13 ->  14 ->  15 -> 0
Distance of the route: 15km

Cost of travel: 500INR
Maximum of the route distances: 15km
Route for vehicle 0:
 0 ->  1 ->  2 ->  3 ->  4 ->  5 ->  6 ->  7 ->  8 ->  9 ->  10 ->  11 ->
12 ->  13 ->  14 ->  15 ->  16 ->  17 ->  18 ->  19 ->  20 ->  21 ->  22 -
>  23 ->  24 ->  25 ->  26 ->  27 ->  28 ->  29 ->  30 ->  31 -> 0
Distance of the route: 31km

Cost of travel: 500INR
Maximum of the route distances: 31km
Route for vehicle 0:
 0 ->  1 ->  2 ->  3 ->  4 ->  5 ->  6 -> 0
Distance of the route: 6km

Cost of travel: 500INR
Maximum of the route distances: 6km
Route for vehicle 0:
 0 ->  17 ->  18 ->  19 ->  6 ->  7 ->  8 ->  9 ->  10 ->  11 ->  12 ->  1
3 ->  14 ->  15 ->  16 ->  1 ->  2 ->  3 ->  4 ->  5 ->  20 ->  21 ->  22
->  23 ->  24 -> 0
Distance of the route: 64km

Cost of travel: 500INR
Maximum of the route distances: 64km
Route for vehicle 0:
 0 ->  1 ->  2 ->  3 ->  4 ->  5 ->  6 ->  7 ->  8 ->  9 ->  10 -> 0
Distance of the route: 10km

Cost of travel: 500INR
Maximum of the route distances: 10km
Route for vehicle 0:
 0 ->  1 ->  2 ->  3 ->  4 ->  5 ->  6 ->  7 ->  8 -> 0
Distance of the route: 8km

Cost of travel: 500INR
Maximum of the route distances: 8km
Route for vehicle 0:
 0 ->  1 -> 0
Distance of the route: 1km

Cost of travel: 500INR
```

```
Maximum of the route distances: 1km
```

Appendix:-

create_gdf

```
def create_gdf(df, x='latitude', y='longitude'):
    return gpd.GeoDataFrame(df,
    geometry=gpd.points_from_xy(df[y], df[x]),
            crs={'init':'epsg:4326'})
stations_gdf = create_gdf(stations)
points_gdf = create_gdf(points)
```

calculate_nearest

```
def calculate_nearest(row, destination, val, col='geometry'):
    # 1 - create unary union
    dest_unary = destination['geometry'].unary_union
    # 2 - find closest point
    nearest_geom = nearest_points(row[col], dest_unary)
    # 3 - Find the corresponding geom
    match_geom = destination.loc[destination.geometry
             == nearest_geom[1]]
    # 4 - get the corresponding value
    match_value = match_geom[val].to_numpy()[0]
    return match_value
```

calculate_cost

```
## function to calculate cost for route
def calculate_cost(distance,average_speed):
    cost = []
    if distance < 100:
        cost = 500
    elif distance > 100 and distance/average_speed < 6 :
        cost = 500 + (distance%100)*10
    else:
        cost = 500 + (distance%100)*10 + 50
    return cost
```

create_data_model

```
#function to stores the data for the problem.
def create_data_model(num):
    data = {}
    data['distance_matrix'] = num
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data
```

print_solution

```
#function to print the solution for the problem.
def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
    max_route_distance = 0
    average_speed = 30
    total_distance = 0
    for vehicle_id in range(data['num_vehicles']):
```

```
        index = routing.Start(vehicle_id)
        plan_output = 'Route for vehicle {}:\n'.format(vehicle_id)
        route_distance = 0
        while not routing.IsEnd(index):
            plan_output += ' {} -> '.format(manager.IndexToNode(index))
            previous_index = index
            index = solution.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(
                previous_index, index, vehicle_id)
        plan_output += '{}\n'.format(manager.IndexToNode(index))
        plan_output += 'Distance of the route: {}km\n'.format(route_distance)
        print(plan_output)
        total_distance += route_distance
        print('Cost of travel: {}INR'.format(calculate_cost(total_distance,30)))
        max_route_distance = max(route_distance, max_route_distance)
    print('Maximum of the route distances: {}km'.format(max_route_distance))
```

cost:

```
def calculate_cost(distance,average_speed):
    cost = []
    if distance < 100:
        cost = 500
    elif distance > 100 and distance/average_speed < 6 :
        cost = 500 + (distance%100)*10
    else:
        cost = 500 + (distance%100)*10 + 50
    return cost
```

main:

```
def main(list_m):
    for i in range(len(list_m)):
        stations_sub = points_gdf[points_gdf['nearest_station'] == list_m[i]]
        mn = pd.DataFrame()
        mn['latitude']=stations_sub['latitude']
        mn['longitude']=stations_sub['longitude']
        temp = stations.loc[stations['Category'] == list_m[i]]
        new_row = pd.DataFrame({'latitude':temp.iloc[0]['latitude'], 'longitude':temp.iloc[0]['longitude']},index =[1])
        mn = pd.concat([new_row, mn]).reset_index(drop = True)
        aa=pd.DataFrame(dist.pairwise(mn[['latitude','longitude']].to_numpy())*6373, index=mn.index,
columns=mn.index)
        # Instantiate the data problem.
        num = aa.reset_index().values
        number_of_trucks = 1 #set the number of trucks to 1
        data = create_data_model(num,number_of_trucks)
        # Create the routing index manager.
        manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),data['num_vehicles'], data['depot'])

        # Create Routing Model.
        routing = pywrapcp.RoutingModel(manager)

        # Create and register a transit callback.
        def distance_callback(from_index, to_index):
        # Convert from routing variable Index to distance matrix NodeIndex.
            from_node = manager.IndexToNode(from_index)
```

```
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]

transit_callback_index = routing.RegisterTransitCallback(distance_callback)

# Define cost of each arc.
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

# Add Distance constraint.
dimension_name = 'Distance'
routing.AddDimension(
transit_callback_index,
0,  # no slack
120,  # vehicle maximum travel distance -> . from travel time of 4 hrs * cruise set speed 30km/hr
True,  # start cumul to zero
dimension_name)
distance_dimension = routing.GetDimensionOrDie(dimension_name)

# Setting first solution heuristic.
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

# Solve the problem.
solution = routing.SolveWithParameters(search_parameters)

# Print solution on console.
if solution:
    print_solution(data, manager, routing, solution)
```