Traveling salesperson Problem between cities

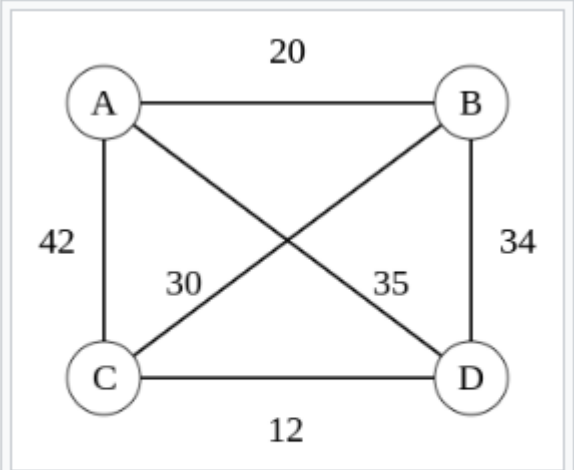
Problem: -

Given a list of cities and the distance between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

Cities	New york	Los angels	Chicago	Minneapolis	Denver	Dallas	Seattle	Boston	San Fransisco	St. Louis	Houston	Phoenix	Salt Lake City
New york	0	2451	713	1018	1631	1374	2408	213	2571	875	1420	2145	1972
Los angels	2451	0	1745	1524	831	1240	959	2596	403	1589	1374	357	579
Chicago	713	1745	0	355	920	803	1737	851	1858	262	940	1453	1260
Minneapolis	1018	1524	355	0	700	862	1395	1123	1584	466	1056	1280	987
Denver	1631	831	920	700	0	663	1021	1769	949	796	879	586	371
Dallas	1374	1240	803	862	663	0	1681	1551	1765	547	225	887	999
Seattle	2408	959	1737	1395	1021	1681	0	2493	678	1724	1891	1114	701
Boston	213	2596	851	1123	1769	1551	2493	0	2699	1038	1605	2300	2099
San Fransisco	2571	403	1858	1584	949	1765	678	2699	0	1744	1645	653	600
St. Louis	875	1589	262	466	796	547	1724	1038	1744	0	679	1272	1162
Houston	1420	1374	940	1056	879	225	1891	1605	1645	679	0	1017	1200
Phoenix	2145	357	1453	1280	586	887	1114	2300	653	1272	1017	0	504
Salt Lake City	1972	579	1260	987	371	999	701	2099	600	1162	1200	504	0

Description: -

TSP was a famous problem since the 18th century and was first mathematically formulated in the 19th century for a school bus routing problem. Since then, it is in circulation.



Symmetric TSP with four cities

We solve this problem using Google or tools in Python.

Code: -

We import all the required libraries.

```
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
import pandas as pd
from IPython.display import display
import folium
from geopy.geocoders import Nominatim
```

- we created a function create_data_model() which is used to store the data for a particular problem. The data for this problem is stored in a dictionary called data.
- The distance_matrix key in this dictionary is a list of lists that represents a distance matrix between various locations. The other keys in the dictionary are num_vehicles, which specifies the number of vehicles available for the problem, and depot, which specifies the index of the starting location.
- The last line of the function **display(pd.DataFrame(data['distance_matrix']).style.background_gradient(cmap='gray'))** is an attempt to display the distance matrix as a styled table. However, it should be noted that this line will never be executed as it is written after the return statement.

```
def create_data_model():
    """Stores the data for the problem."""
    data = {}
    data['distance_matrix'] = [
        [0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145,
1972],
        [2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357,
579],
        [713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453,
1260],
        [1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280,
987],
        [1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],
        [1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887,
999],
        [2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114,
701],
        [213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300,
2099],
        [2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653,
600],
        [875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272,
1162],
        [1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017,
1200],
        [2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0,
504],
        [1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504,
0],
    ]  # yapf: disable
    data['num_vehicles'] = 1
```

```
    data['depot'] = 0
    return data

display(pd.DataFrame(data['distance_matrix']).style.background_gradient(cma
p='gray'))
```

- **print_solution()** that takes in three parameters: manager, routing, and solution. The manager parameter is an object that keeps track of the indices of the locations in the distance matrix. The routing parameter is an object that represents the vehicle routing problem with time windows. The solution parameter is an object that represents the solution to the vehicle routing problem.
- The function prints the objective value of the solution, which is the total distance traveled by the vehicle.This is obtained by calling the ObjectiveValue method on the solution object. The function then initializes a variable called index to the starting location index, and another variable called plan_output to the string "Route for vehicle 0:\n". It also initializes a variable called route_distance to 0.
- Next, the function creates a list of city names corresponding to the indices of the locations in the distance matrix. This list is used later to print the names of the cities instead of their indices. The function then enters a loop that continues until the end of the route is reached. Inside the loop, the function adds the name of the current city to plan_output and updates the index variable to the index of the next location in the route.
- The route_distance variable is also updated to include the distance traveled between the current and next cities. After the loop is finished, the function adds the final destination city to plan_output and prints the final string, which includes the route taken by the vehicle and the total distance traveled. The plan_output string is also returned.

```
def print_solution(manager, routing, solution):
    """Prints solution on console."""
    print('Objective: {} miles'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    city_names = ['New York','Los
Angeles','Chicago','Minneapolis','Denver','Dallas',
                  'Seattle','Boston','San Francisco','St.
Louis','Houston','Phoenix','Salt Lake City']
    while not routing.IsEnd(index):
        plan_output += ' {} -
>'.format(city_names[manager.IndexToNode(index)])
        previous_index = index
```

```
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index,
index, 0)
    plan_output += ' {}\n'.format(city_names[manager.IndexToNode(index)])
    print(plan_output)
    plan_output += 'Route distance: {}miles\n'.format(route_distance)
```

The main function main() does the following:

- Calls create_data_model() function to create a dictionary object data that contains the distance matrix, number of vehicles, and the depot location.
- Creates a RoutingIndexManager object that manages the indices of nodes in the TSP.
- Creates a RoutingModel object that defines the optimization problem for the TSP.
- Defines a distance_callback function that returns the distance between two nodes.
- Registers the distance_callback function as the transit callback function for the TSP.
- Sets the arc cost evaluator for all vehicles to the transit_callback_index.
- Defines the first solution strategy as the path cheapest arc.
- Solves the TSP using the SolveWithParameters() method of the RoutingModel object.
- If a solution is found, it calls the print_solution() function to print the solution on the console.
- The print_solution() function takes the manager, routing, and solution objects as arguments and prints the objective value, the route for the vehicle, and the route distance on the console.

```
def main():
    """Entry point of the program."""
    # Instantiate the data problem.
    data = create_data_model()

    # Create the routing index manager.
    manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
                                           data['num_vehicles'],
data['depot'])

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


    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""
        # 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)

    # 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(manager, routing, solution)

if __name__ == '__main__':
    main()
```

Visualizing the result on a map using Folium.

- We first define a list of city names, then use Geopy to get each city's coordinates (latitude and longitude). We then create a map object using Folium, add a marker for each city, and display the map.
- In the script then defines the TSP problem data using the create_data_model() function, creates a routing index manager and a routing model using OR-Tools, and defines a callback function to calculate the distance between two nodes (i.e., cities).
- It then sets the cost evaluator for each arc using the callback function, sets the first solution strategy to PATH_CHEAPEST_ARC, and solves the TSP problem using OR-Tools.
- If a solution is found, the script calls the **print_solution()** function to print the route (i.e., the order in which the cities should be visited) on the console. It then creates a new Folium map, adds markers for each city, and adds a red polyline to connect the cities in the order specified by the solution. Finally, it displays the map with the route.

```
•   import folium
    from geopy.geocoders import Nominatim

    # Define the city names
    #city_names = ['oklahoma', 'dallas', 'new york', 'austin', 'moore']
    city_names = ['New York','Los
    Angeles','Chicago','Minneapolis','Denver','Dallas',
                      'Seattle','Boston','San Francisco','St.
```

```
Louis','Houston','Phoenix','Salt Lake City']
# Create a geolocator object
# Create a geolocator object
geolocator = Nominatim(user_agent="my_application")

# Get the coordinates of each city
locations = {}
for city in city_names:
    location = geolocator.geocode(city)
    locations[city] = (location.latitude, location.longitude)

# Plot the map
m = folium.Map(location=list(locations.values())[0], zoom_start=5)

# Add markers for each city
for city, loc in locations.items():
    folium.Marker(location=loc, tooltip=city).add_to(m)

def print_solution(manager, routing, solution):
    index = routing.Start(0)
    route = []
    while not routing.IsEnd(index):
        route.append(city_names[manager.IndexToNode(index)])
        index = solution.Value(routing.NextVar(index))
    route.append(city_names[manager.IndexToNode(index)])
    return route
def main():
    data = create_data_model()
    manager =
pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
data['num_vehicles'], data['depot'])
    routing = pywrapcp.RoutingModel(manager)

    def distance_callback(from_index, to_index):
        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)
    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy =
routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    solution = routing.SolveWithParameters(search_parameters)
    if solution:
        route = print_solution(manager, routing, solution)

        # Plot the map
        m = folium.Map(location=list(locations.values())[0],
zoom_start=5)

        # Add markers for each city
        for city, loc in locations.items():
            folium.Marker(location=loc, tooltip=city).add_to(m)

        # Add the route
        route_coords = [locations[city] for city in route]
        folium.PolyLine(route_coords, color="red", weight=2.5,
opacity=1).add_to(m)
```

```
        # Show the map
        display(m)

if __name__ == '__main__':
    main()
```

Result: -

Distance Matrix

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	2451	713	1018	1631	1374	2408	213	2571	875	1420	2145	1972
1	2451	0	1745	1524	831	1240	959	2596	403	1589	1374	357	579
2	713	1745	0	355	920	803	1737	851	1858	262	940	1453	1260
3	1018	1524	355	0	700	862	1395	1123	1584	466	1056	1280	987
4	1631	831	920	700	0	663	1021	1769	949	796	879	586	371
5	1374	1240	803	862	663	0	1681	1551	1765	547	225	887	999
6	2408	959	1737	1395	1021	1681	0	2493	678	1724	1891	1114	701
7	213	2596	851	1123	1769	1551	2493	0	2699	1038	1605	2300	2099
8	2571	403	1858	1584	949	1765	678	2699	0	1744	1645	653	600
9	875	1589	262	466	796	547	1724	1038	1744	0	679	1272	1162
10	1420	1374	940	1056	879	225	1891	1605	1645	679	0	1017	1200
11	2145	357	1453	1280	586	887	1114	2300	653	1272	1017	0	504
12	1972	579	1260	987	371	999	701	2099	600	1162	1200	504	0

The distance matrix is the NxN matrix with the distance between each and every city.

Objective and Route: -

```
Objective: 7293 miles
Route for vehicle 0:
 New York -> Boston -> Chicago -> Minneapolis -> Denver -> Salt Lake City -> Seattle -> San Francisco -> Los Angeles -> Phoenix
-> Houston -> Dallas -> St. Louis -> New York
```

We got the objective value as 7293 miles and a route map to cover the cities and return to the original destination.

Visualization: -

We have plotted the route in the map graph and visualized as below.

```
Route for vehicle 0:
 New York -> Boston -> Chicago -> Minneapolis -> Denver -> Salt Lake City -> Seattle -> San Francisco -> Los Angeles -> Phoenix
-> Houston -> Dallas -> St. Louis -> New York
```