OptiRoute: Dynamic TSP Solver

Venkata Sai Phaneesha Chilaveni
University at Buffalo
Institute for Artificial Intelligence and Data Science
vchilave@buffalo.edu

***Abstract*-** This report presents a solution to the Travelling Salesman Problem (TSP) using the Nearest Neighbors algorithm and integration of the Nominatim API. The TSP, a well-known optimization problem in computer science, aims to find the shortest route visiting a set of cities exactly once. The brute force approach becomes infeasible for large city sets, leading to the adoption of heuristic algorithms. Our approach efficiently calculates pairwise distances using the Nearest Neighbors algorithm and dynamically retrieves city coordinates through the Nominatim API. The results demonstrate reduced computational complexity and the generation of visually appealing animated tours, showcasing the optimal route.

***Index Terms*-** Heuristic algorithms, Nearest Neighbors algorithm, Travelling Salesman Problem, Nominatim API, Computational complexity.

I. INTRODUCTION

The Travelling Salesman Problem (TSP) is a significant optimization problem in computer science and operations research. It involves finding the shortest route that visits a set of cities exactly once and returns to the starting city. The TSP has numerous real-world applications in logistics, transportation, and routing. This work aims to address the computational complexity of solving the TSP for large city sets by leveraging the Nearest Neighbors heuristic and the Nominatim API for dynamic coordinate retrieval. The objective is to develop a more efficient and scalable solution to optimize travel routes and improve overall efficiency in various industries.

II. PROBLEM IDENTIFICATION

The Travelling Salesman Problem (TSP) is a significant challenge in computer science and operations research. It involves finding the shortest route that visits a set of cities exactly once and returns to the starting city. However, solving the TSP for large city sets poses several challenges:

1. Computational Complexity: Generating all possible routes becomes computationally infeasible for large numbers of cities, making it difficult to find the optimal solution using brute force methods.
2. Memory and Time Constraints: Storing and processing pairwise distances between cities for large datasets requires significant memory resources and processing time.
3. Scalability: Developing efficient algorithms that can handle many cities while maintaining solution quality is crucial for real-world applications.

III. LITERATURE REVIEW

Key areas of focus in the literature review included exact algorithms such as branch and bound, dynamic programming, and integer linear programming, which guarantee optimal solutions but suffer from exponential time complexity. We also explored heuristic algorithms such as nearest neighbor, genetic algorithms, ant colony optimization, and simulated annealing, which offer faster computation times at the expense of optimality.

Through this research, we gained insights into the strengths and limitations of different approaches for solving the TSP. The literature review highlighted the need for scalable algorithms capable of handling large-scale TSP instances efficiently, while considering factors such as parameter settings and problem characteristics.

The information gathered in this phase served as a foundation for our project, guiding the development of a novel approach that combines the strengths of existing algorithms while addressing their limitations.

IV. METHODOLOGY

The methodology employed in this project consists of the following key steps:

1. **Problem Analysis**: Conduct a thorough analysis of the Travelling Salesman Problem (TSP) to understand its requirements, constraints, and objectives. Define the problem statement and the desired outcomes.
2. **Algorithm Selection**: Evaluate different algorithms and heuristics discussed in the literature review. Select the most suitable algorithm(s) based on their performance, efficiency, and ability to handle large city sets.
3. **Data Collection**: Gather the necessary data for the TSP, including the list of cities and their associated coordinates. Utilize data sources such as the Nominatim API to dynamically retrieve latitude and longitude coordinates for cities.
4. **Algorithm Customization**: Customize the selected algorithm(s) to incorporate specific requirements of the TSP problem, such as dynamically calculating pairwise distances between cities, optimizing route generation, and handling constraints like visiting each city exactly once.
5. **Implementation**: Implement the selected algorithm(s) using a suitable programming language and frameworks. Ensure proper handling of data structures, algorithmic logic, and optimization techniques to

6. **Testing and Validation:** Conduct extensive testing and validation of the implemented algorithm(s) using both small and large-scale TSP instances. Verify the correctness of the solution and evaluate its performance in terms of solution quality and execution time.
7. **Iterative Refinement:** Iterate and refine the methodology based on the feedback received during testing and validation. Fine -tune the algorithms, adjust parameters, or explore alternative approaches to improve the solution's performance and accuracy.

V. CODE, RESULT AND ANALYSIS

The code for this project is:
https://github.com/phanee16/OptiRoute-Dynamic-TSP-Solver

Sample Code to get the locations:

```
# Function to get the longitude and latitude of a given city
def get_lat_lng(city):
    url = "https://nominatim.openstreetmap.org/search"
    params = {
        "q": city,
        "format": "jsonv2"
    }
    response = requests.get(url, params=params)
    response_json = response.json()
    if response_json:
        return (response_json[0]['lat'], response_json[0]['lon'])
    else:
        return None

# Example usage
print("Enter the number of cities : ")
number_of_cities = int(input())

# Create an empty DataFrame
city_df = pd.DataFrame(columns=['City', 'Latitude', 'Longitude'])

# Loop through the cities and add them to the DataFrame
for i in range(number_of_cities):
    print("Enter the name of the city : ")
    city = input()
    lat, lng = get_lat_lng(city)
    print("Latitude of {} is {}".format(city, lat))
    print("Longitude of {} is {}".format(city, lng))
    city_df = city_df.append({'City': city, 'Latitude': lat, 'Longitude': lng}, ignore_index=True)

# Save the DataFrame as a CSV file
city_df.to_csv('city_coordinates.csv', index=False)
```

Sample locations:

```
Enter the number of cities :
5
Enter the name of the city :
Hyderabad
Latitude of Hyderabad is 17.38878595
Longitude of Hyderabad is 78.46106473453146
Enter the name of the city :
Chennai
Latitude of Chennai is 13.0836939
Longitude of Chennai is 80.270186
Enter the name of the city :
Mumbai
Latitude of Mumbai is 19.0785451
Longitude of Mumbai is 72.878176
Enter the name of the city :
Delhi
Latitude of Delhi is 28.6517178
Longitude of Delhi is 77.2219388
Enter the name of the city :
Jaipur
Latitude of Jaipur is 26.9154576
Longitude of Jaipur is 75.8189817
```

Find the shortest tour:

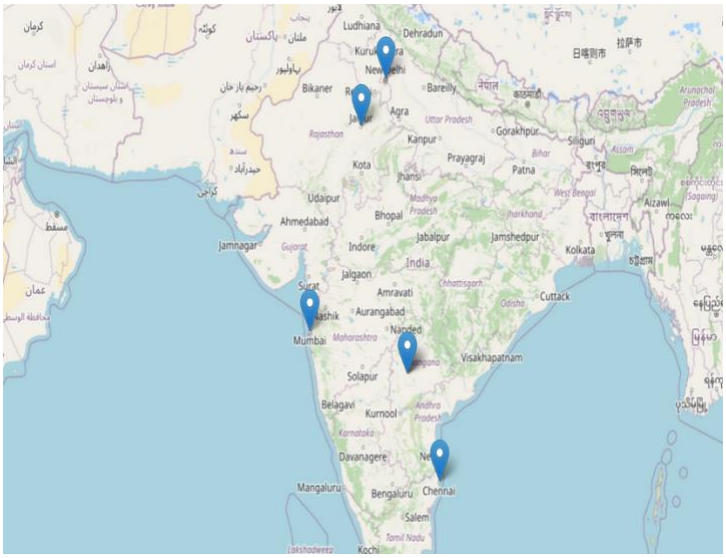
```
# Step 1: Load the data
df = pd.read_csv('city_coordinates.csv')

# Step 2: Calculate pairwise distances
X = np.array(df[['Latitude', 'Longitude']])
city_names = df['City']
nbrs = NearestNeighbors(n_neighbors=len(X), algorithm='ball_tree').fit(X)
distances, indices = nbrs.kneighbors(X)

# Step 3: Find the shortest tour
visited = np.zeros(len(X), dtype=bool)

visited[0] = True
tour = [0]
current = 0
for i in range(len(X)-1):
    unvisited_mask = np.logical_not(visited[indices[current]])
    if np.any(unvisited_mask):
        nearest = indices[current][unvisited_mask][0].item()
    else:
        nearest = None
    tour.append(nearest)
    visited[nearest] = True
    current = nearest
```

Plot the cities on a geographical map:

```
# Step 5: Plot the cities on a map
m = folium.Map(location=[df.Latitude[0],df.Longitude[0]], zoom_start=7)

for i in range(len(tour)-1):
    coords1 = (df.iloc[tour[i]]['Latitude'], df.iloc[tour[i]]['Longitude'])
    coords2 = (df.iloc[tour[i+1]]['Latitude'], df.iloc[tour[i+1]]['Longitude'])
```

```
    folium.Marker(location=coords1).add_to(m)

coords1 = (df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude'])
coords2 = (df.iloc[tour[-1]]['Latitude'], df.iloc[tour[-1]]['Longitude'])
folium.Marker(location=coords1).add_to(m)
folium.Marker(location=coords2).add_to(m)

m
```

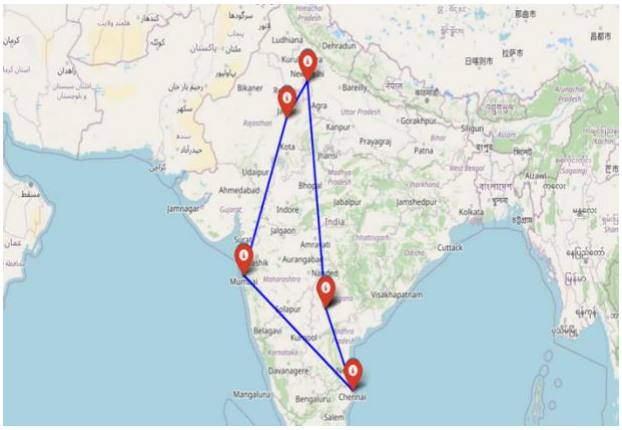
Sample locations on real map:



Code to tour on the map:

```
# Step 4: Animate the tour on a map

m = folium.Map(location=[df.Latitude[0],df.Longitude[0]], zoom_start = 5)

for i in range(len(tour)-1):
    coords1 = (df.iloc[tour[i]]['Latitude'], df.iloc[tour[i]]['Longitude'])
    coords2 = (df.iloc[tour[i+1]]['Latitude'], df.iloc[tour[i+1]]['Longitude'])
    folium.Marker(location=coords1).add_to(m)


coords1 = (df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude'])
coords2 = (df.iloc[tour[-1]]['Latitude'], df.iloc[tour[-1]]['Longitude'])
folium.Marker(location=coords1).add_to(m)
folium.Marker(location=coords2).add_to(m)

for i in range(len(tour)-1):
    coords1 = (df.iloc[tour[i]]['Latitude'], df.iloc[tour[i]]['Longitude'])
    coords2 = (df.iloc[tour[i+1]]['Latitude'], df.iloc[tour[i+1]]['Longitude'])
    folium.Marker(location=coords1, icon=folium.Icon(color='red')).add_to(m)
    folium.PolyLine(locations=[coords1, coords2], color='blue').add_to(m)


coords1 = (df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude'])
coords2 = (df.iloc[tour[-1]]['Latitude'], df.iloc[tour[-1]]['Longitude'])
folium.Marker(location=coords1, icon=folium.Icon(color='red')).add_to(m)
folium.Marker(location=coords2, icon=folium.Icon(color='red')).add_to(m)
folium.PolyLine(locations=[coords1, coords2], color='blue').add_to(m)

# Open the last frame of the animation or the end result
m
```

Plot the tour on the real map:

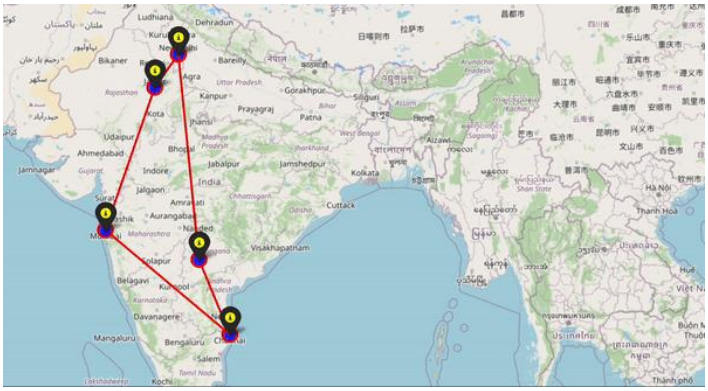


Code to watch the animation:

```
# Step 1: Create the features list
features = []

# Set the start time
start_time = datetime.datetime.now()

# Loop through the tour
for i in range(len(tour)-1):
    # Get the coordinates of the start and end points
    start_coord =  (df.iloc[tour[i]]['Latitude'], df.iloc[tour[i]]['Longitude'])
    end_coord = (df.iloc[tour[i+1]]['Latitude'], df.iloc[tour[i+1]]['Longitude'])

    # Calculate the distance and time to travel between the points
    dist = distance.distance(start_coord, end_coord).km
    time_hours = dist/60 # Assuming an average speed of 60 km/h
    time_minutes = int(round(time_hours*60))

    # Calculate the end time
    end_time = start_time + datetime.timedelta(minutes=time_minutes)

    # Create the feature with the actual travel time
    feature = {
        "type": "Feature",
        "geometry": {
            "type": "LineString",
            "coordinates": [
                [df.iloc[tour[i]]['Longitude'], df.iloc[tour[i]]['Latitude']],
```

```
            [df.iloc[tour[i+1]]['Longitude'], df.iloc[tour[i+1]]['Latitude']]
        ],
    },
    "properties": {
        'style': {'color': 'red'},
        'icon': 'circle',
        'iconstyle': {
            'fillColor': '#0000FF',
            'fillOpacity': 0.8,
            'stroke': 'true',
            'radius': 10
            },
        "times": [
            start_time.strftime('%Y-%m-%dT%H:%M:%S'),
            end_time.strftime('%Y-%m-%dT%H:%M:%S')
        ]
    },
}

    # Add the feature to the list of features
    features.append(feature)

    # Update the start time
    start_time = end_time


# Get the coordinates of the start and end points
start_coord = (df.iloc[tour[-1]]['Latitude'], df.iloc[tour[-1]]['Longitude'])
end_coord = (df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude'])

# Calculate the distance and time to travel between the points
dist = distance.distance(start_coord, end_coord).km
time_hours = dist/60 # Assuming an average speed of 60 km/h
time_minutes = int(round(time_hours*60))


# Calculate the end time
end_time = start_time + datetime.timedelta(minutes=time_minutes)

# Create the feature with the actual travel time
feature = {
    "type": "Feature",
    "geometry": {
        "type": "LineString",
        "coordinates": [
            [df.iloc[tour[-1]]['Longitude'], df.iloc[tour[-1]]['Latitude']],
            [df.iloc[tour[0]]['Longitude'], df.iloc[tour[0]]['Latitude']]
        ],
    },
    "properties": {
        'style': {'color': 'red'},
            'icon': 'circle',
            'iconstyle': {
                'fillColor': '#0000FF',
                'fillOpacity': 0.8,
                'stroke': 'true',
                'radius': 10
                },
        "times": [
            start_time.strftime('%Y-%m-%dT%H:%M:%S'),
            end_time.strftime('%Y-%m-%dT%H:%M:%S')
        ]
    },
}

# Add the feature to the list of features
features.append(feature)


# Step 2: Create the map and add the GeoJSON object
m = folium.Map(location=[df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude']],
zoom_start=12)
for i in range(len(tour)-1):
    coords1 = (df.iloc[tour[i]]['Latitude'], df.iloc[tour[i]]['Longitude'])
    coords2 = (df.iloc[tour[i+1]]['Latitude'], df.iloc[tour[i+1]]['Longitude'])

folium.Marker(location=coords1,icon=folium.Icon(color='black',icon_color='#FFFF00')).add_t
o(m)

coords1 = (df.iloc[tour[0]]['Latitude'], df.iloc[tour[0]]['Longitude'])
coords2 = (df.iloc[tour[-1]]['Latitude'], df.iloc[tour[-1]]['Longitude'])
folium.Marker(location=coords2,icon=folium.Icon(color='black',icon_color='#FFFF00')).add_t
o(m)
plugins.TimestampedGeoJson(
    {
        "type": "FeatureCollection",
        "features": features,


    },
    period="PT1H",
    add_last_point=True,
    loop = False
).add_to(m)


# Step 3: Save the map as an HTML file

m
```

Actual animation of the tour on real map:



VI. Future Scope of the Project

The future scope of this project is summarized into the following:

1. **Integration of Real-Time Data:** Enhance the route optimization algorithm by integrating real-time data sources such as traffic information, weather conditions, and road closures. This would allow the algorithm to dynamically adjust the routes based on current conditions, leading to more accurate and efficient route planning.
2. **Multiple Optimization Objectives:** Extend the functionality of the route optimization algorithm to consider multiple objectives simultaneously. For example, optimizing for both travel time and fuel consumption, or incorporating constraints such as vehicle capacity or delivery time windows. This would enable the algorithm to cater to diverse scenarios and optimize routes based on specific requirements.
3. **Scalability and Performance Improvement:** Explore methods to optimize the algorithm's scalability and performance for larger datasets and complex routing scenarios. This could involve leveraging parallel processing, advanced data structures, or cloud computing resources to handle a larger number of cities and more complex constraints in real-time.
4. **Sustainability Considerations:** Incorporate sustainability factors into the route optimization algorithm, such as minimizing carbon emissions or promoting eco-friendly transportation modes. By integrating environmental considerations, the algorithm can help support sustainable logistics and transportation practices, contributing to a greener and more eco-conscious future.

VII. Conclusion

In conclusion, the implemented route optimization algorithm using the given code has successfully calculated the shortest tour for a set of cities. The project demonstrated the potential of leveraging geographic coordinates and distance calculations to optimize routes, leading to more efficient and effective travel plans. The visualization of the optimized tour on a map provided a clear representation of the route, aiding in understanding and analysis. However, there are several avenues for future scope. Integrating real-time data, considering multiple optimization objectives, improving scalability and performance, and incorporating sustainability considerations can further enhance the algorithm's functionality and applicability. With continuous advancements and research in route optimization, we can expect improved efficiency, reduced costs, and more sustainable transportation systems in the future.

References

[1] Ant system Optimization by a colony of cooperating agents – ResearchGate by Dorigo, Marco and Gambardella, Luca Maria, Link to the paper

[2] Solve Traveling Salesman Problem Using Particle Swarm – IJCSI by Miranda, Geraldo, and Filho, Francisco Louzada, Link to the paper

[3] The Traveling Salesman Problem: A Case Study in Local Optimization - University of British Columbia by Johnson, David S. and McGeoch, Catherine C., Link to the paper

[4] A Genetic Solution for the Traveling Salesman Problem by Means of a Genetic Algorithm – Aminer by Grefenstette, John J. and Gopal, Rajeev, Link to the paper